Advanced Data Structures

Summary

January 9, 2020

Red-black trees

Red-back trees are binary search trees where every node has an additional field: a color which is either **red** or **black**. A suitable `C++` class to represent the nodes of a red-black tree with keys of integer type is

```
struct RBNode {
   int key;                  // key
   ...                       // satellite data (can be anything)
   RBNode* left;             // pointer to left child
   RBNode* right;            // pointer to right child
   RBNode* p;                // pointer to parent node
   enum color {RED, BLACK };
   color col;                // color
};
```

A red-black tree is a binary search tree ⇒ it most satisfy the conditions of a binary search tree:

- Every node has satellite data and a unique key.
- The keys belong to a totally ordered set of values (e.g., strings or integers).
- The keys are distributed in a special way: For every node, its key is larger then the keys of the nodes in its left subtree, and larger than the keys of the nodes in its right subtree.

In addition, a red-black tree must have the following red-black properties: .

1. Every node is either red or black.
2. The root node is black.
3. The leaves are the `NULL` nodes, and they are considered to be black.
4. The children of a red node are black.
5. Every path from a node x to a descendant leaf contains the same number of black nodes.

The number of black nodes on a path from a node x of a red-black tree to a leaf, excluding x, is called the **black height** of x in the tree.

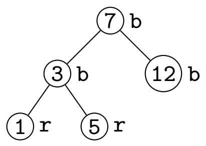
Questions and exercises

E1. Indicate the main red-black properties of a red-black tree.

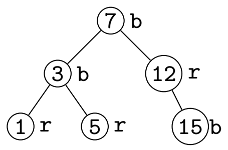
E2. Indicate the derived properties of a red-black tree.

E3. What are the main operations supported by red-black trees, and what is their runtime complexity? What is the advantage of using red-back trees instead of binary search trees?

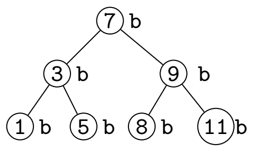
E4. The following pictures illustrate binary trees whose nodes have color red or black, and keys which are integer values. (The color of a node is indicated with `r` or `b` besides the node of the tree.)



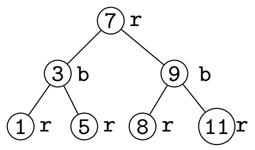
T_1



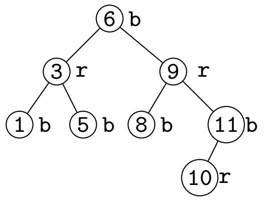
T_2



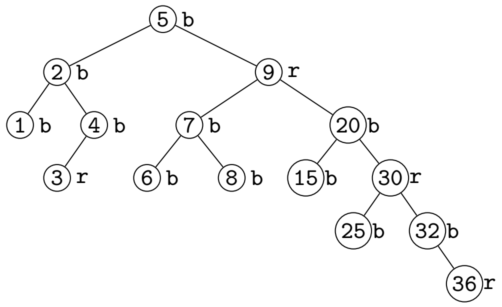
T_3



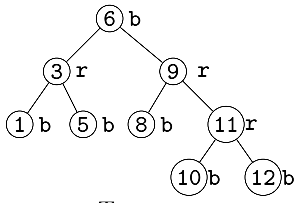
T_4



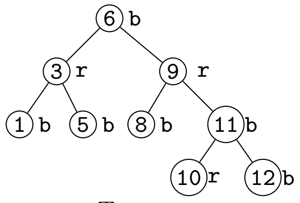
T_5



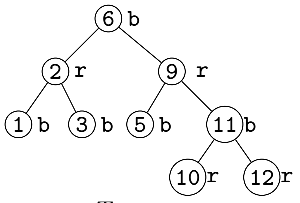
T_6



T_7



T_8



T_9

Indicate which of these trees are red-black trees, and their red-black heights. For those which are not red-black trees, indicate a reason why not.

E5. Indicate the red-black tree produced by inserting a node with key `34` in the tree T_6 of Exercise E4.

E6. Indicate the red-black tree produced by removing the node with key `20` from the tree T_6 of Exercise E4.

E7. Suppose `r` is a pointer to the root of a binary search tree whose nodes are instances of the class

```
struct Node {
    int key;
    Node* p;     // pointer to parent
    Node* left;  // pointer to left child
    Node* right; // pointer to right child
}
```

Write down an implementation of the function

```
Node* tree_maximum(Node* r)
```

which returns a pointer to the node with maximum key from the tree with root `r`. What is the worst runtime complexity of this operation, if the binary search tree has n nodes?

E8. Suppose `x` is a pointer to a node (not necessarily the root) of a binary search tree whose nodes are instances of the class `Node` mentioned before. Write down an implementation of the function

```
Node* maximum(Node* x)
```

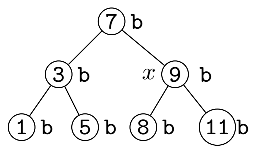
which returns a pointer to the node with maximum key from the tree to which contains the node pointed by `x`. What is the worst runtime complexity of this operation, if the binary search tree has n nodes?

E9. Write down an efficient implementation of the function

```
int bh(RBNode* x)
```

which takes as input a pointer `x` to the root of a red-black tree, and returns the black-height of the red-black tree with root referred by `x`. Indicate the runtime complexity of your implementation of function `bh()`.

E10. Indicate the results of LeftRotate(T, x) and RightRotate(T, x), where T is the tree with node x illustrated below:



Answers

E1. There are five main red-black properties (see lecture notes).

E2. The derived properties of a red-black tree are:

(a) For every node n, $bh(n) \geq h(n)/2$

(b) The subtree rooted at any node x contains $\geq 2 \cdot bh(x) - 1$ internal nodes.

(c) A red-black tree with n internal nodes has height $\leq 2\log_2(n+1)$.

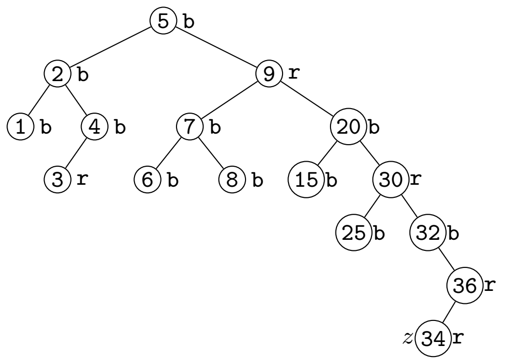
where $bh(x)$ is the black-height of node x, that is, the number of black nodes on any path from x to a leaf, excluding x; and $h(x)$ is the height of the subtree rooted at node x.

E3. The main operations supported by red-black trees are: (1) search an element with an given key, (2) find the element with minimum key, (3) find the element with maximem key, (3) find successor = element with next largest key, (4) find predecessor = element with next smallest key, (5) insert a new element, (6) delete element with a given key.

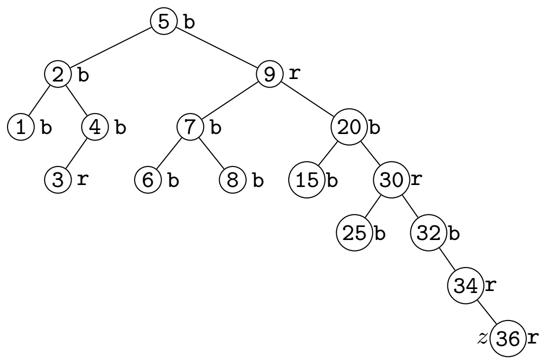
- The runtime complexity of all these operations is $O(n)$ where n is the number of elements in the tree.
- The worst-case runtime complexity of the operations of binary search trees is $O(n)$, and the worst-case runtime complexity of the operations of red-black trees is $O(\log n)$.

E4. The following are red-black trees: T_1 with black-height 2; T_3 with black-height 3; T_5 with black-height 2; and T_6 with black-height 3.
T_2 is not red-black tree because it does not satisfy red-black property 5: the path from root to the left child of `12` (which is `NULL`) has 3 black nodes, and the path from root to the right child of `12` has 3 black nodes. T_4 is not red-black tree because its root is red. T_7 is not red-black tree because red node `9` has red child `11`. T_8 is not red-black tree because it does not satisfy red-black property 5. T_9 is not red-black tree because it is not a binary search tree.

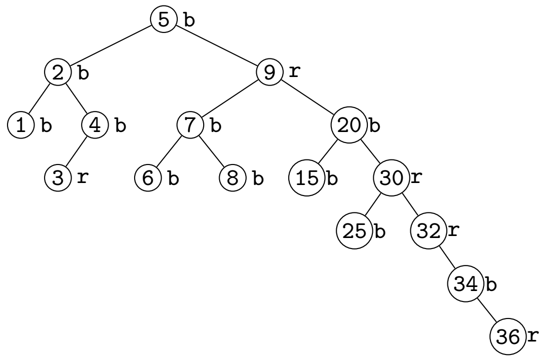
E5. First, we insert node z with key `34` like in a binary search tree, and assign color red to z:



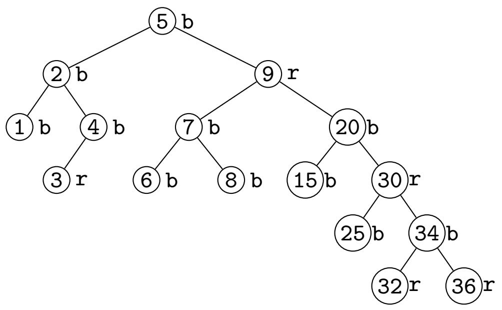
Next, we call RBInsertFixup(z) (see Appendix) because red-black property 4 does not hold: z and $z \to$ `p` are both red. z is left child and its uncle is black $\Rightarrow$ we perform the operations of Case 2 with `right` and `left` exchanged (because $z \to$ `p` $\neq z \to$ `p` $\to$ `p` $\to$ `left`), and obtain



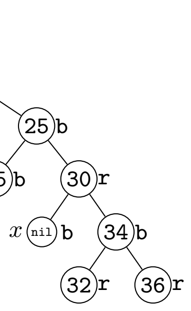
Next, we apply Case 3 with `right` and `left` exchanged. The recolorings of $z \to$ `p` and $z \to$ `p` $\to$ `p` yield



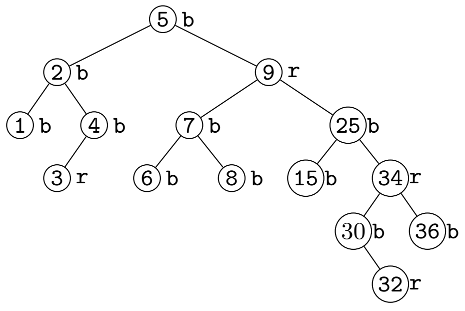
Afterwards, LeftRotate($z \to$ `p` $\to$ `p`) yields the red-black tree



E6. To delete node with key 20 from T_6, we splice out its successor, which is the black node with key 25. Therefore, we call RBDeleteFixup(x) (see Appendix) where x is the sentinel node `nil`.



We apply Case 4 and obtain



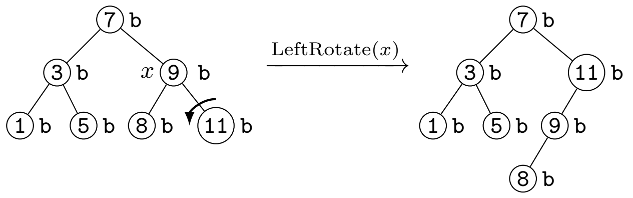
E7.
```
Node* tree_maximum(Node* r) {
    if (r == Nil) return r;
    while (r->right != Nil) r=r->right;
    return r;
}
```

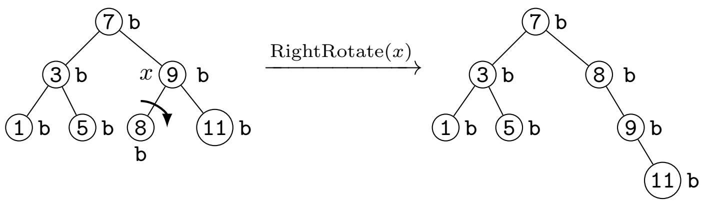
E8. See Appendix.

E9.
```
int bh(RBNode* x) {
    if (x==Nil) return 0;
    return (x->col==RED)?bh(x->left):(bh(x->left)-1);
}
```

The runtime complexity of this implementation is proportional to the depth of the red-black tree referred by `x`, which is $O(\log_2 n)$, where n is the number of nodes in the tree.

E10.





Disjoint-set structures

A disjoint-set data structure is a container for a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$ of non-empty dynamic sets which are disjoint. (Two sets A, B are disjoint if $A \cap B = \emptyset$.)

Each set is identified by a member of the set, called its **representative.**

A disjoint-set data structure fulfils the following requirement:

- ▶ If we ask for the representative of a dynamic set twice without modifying the set, we should get the same answer.

The main operations supported by a disjoint set data structure S are:

1. MakeSet(x): extends S with a new singleton set $\{x\}$. It is assumed that x is not already an element of another set of S.
2. Union(x, y): unites the sets that contain x and y, say S_x and S_y, into a new set that is their union. The sets S_x and S_y can be destroyed.
3. FindSet(x): returns a pointer to the representative element of the set of S which contains element x.

Questions and exercises

E11. Indicate a `C++` structure for the linked-list representation of a disjoint-set structure, together with the meaning of its fields and the implementation its main operations. Consider the set elements to be strings.

E12. Indicate a `C++` structure for the rooted-tree representation of a disjoint-set structure, together with the meanings of its fields and the implementation of its main operations. Consider the set elements to be strings.

E13. Explain how can we use a disjoint set data structure to compute the connected components of an undirected graph $G = (V, E)$.

E14. Explain how can we use a disjoint set data structure to compute a minimum spanning tree (MST) of a connected, undirected, weighted graph.

Answers

E11. In the linked-list representation, every subset of a disjoint set data structure is an instance of the class `LinkedListElem`. For efficiency reasons, it is also useful to maintain two global data structures: (1) a map `tail`, such that `tail[x]` refers to the last element in the linked list representation of the set with representative `x`, and (2) a map `len`, such that `len[x]` refers to the length of the linked list representation of the set with representative `x`.

The class `LinkedListElem` and its operations can be defined as follows.

```
struct LinkedListElem;
using ElPtr = LinkedListElem*;
static map<ElPtr,ElPtr> tail;
static map<ElPtr,int> len;

struct LinkedListElem {
   string id;   // element name
   ElPtr next;  // pointer to next element in its set
   ElPtr repr;  // pointer to its representative

   LinkedListElem(string s) : id(s), next(0) {
      repr = this; tail[this] = this; len[this] = 1;
   }
   static ElPtr MakeSet(string s) { return new LinkedListElem(s); }
   static ElPtr FindSet(ElPtr x) { return x->repr; }
   static void Union(ElPtr x, ElPtr y) {
      ElPtr xSet = FindSet(x);
      ElPtr ySet = FindSet(y);
      if (xSet != ySet) {
         if (len[xSet] < len[ySet]) { // swap xSet with ySet
            ElPtr tmp = xSet;
            xSet = ySet;
            ySet = tmp;
         }
         // find z = the last element of set xSet
         ElPtr z = tail[xSet];
         z->next = ySet;
         // remove ySet as a set representative from tail
         tail[xSet] = tail[ySet];
         len[xSet] += len[ySet];
         tail.erase(ySet);
         len.erase(ySet);
         z = ySet;
         while (z != 0) {
            z->repr = xSet;
            z = z->next;
         }
      }
   }
};
```

E12. In the rooted-tree representation, every subset of a disjoint set data structure is an instance of the class `TreeElem`. The class `TreeElem` and its operations can be defined as follows.

```
struct TreeElem;
using ElPtr = TreeElem*;

static map<string,ElPtr> elem;

struct TreeElem {
   string id; // node identifier
   int rank;  // an upper bound on the height of this tree
   ElPtr p;   // pointer to the parent node

   TreeElem(string s) : id(s), rank(0) {
      p = this;
   }
   static ElPtr MakeSet(string s) {
      return new TreeElem(s);
   }
   static void Link(ElPtr x, ElPtr y) {
      if (x->rank>y->rank)
         y->p=x;
      else {
         x->p=y;
         if (x->rank==y->rank)
            y->rank=y->rank+1;
      }
   }
   static void Union(ElPtr x, ElPtr y) {
      Link(FindSet(x),FindSet(y));
   }
   static ElPtr FindSet(ElPtr x) {
      if (x!=x->p) x->p=FindSet(x->p);
      return x->p;
   }
};
```

Binomial heaps

A binomial heap is an advanced data structure designed to support well the following operations on dynamic sets:

1. makeHeap(): creates and returns a new empty heap.
2. insert(H, x): inserts node referred by x, whose key field has already been filled in, into heap H.
3. minimum(H): returns a pointer to the node with minimum key in heap H.

4. extractMin(H): deletes the node with minimum key from heap H, and returns a pointer to the deleted node.
5. Union($H1, H2$): creates and returns a new heap that contains all the nodes of heaps $H1$ and $H2$. Heaps $H1$ and $H2$ are destroyed by this operation.
6. decreaseKey(x, k): assigns to node referred by x within the heap the new key value k. It is assumed that $k \leq x \to key$.
7. del(H, x): deletes the node referred by x from heap H.

Before defining binomial heaps, we define binomial trees. A **binomial tree** is an element of the set of ordered trees $\{B_k \mid k \in \mathbb{N}\}$ defined recursively as follows:

B_0 consists of a single node.
For any $k > 0$, B_k consists of two binomial trees B_{k-1} that are linked such that one of them has the other one as left child of its root.

A **binomial heap** H is a set of binomial trees that satisfies the following properties:

1. Each binomial tree in H is **heap-ordered**: the key of a node is greater than the key of its parent.
2. There is at most one binomial tree in H whose root has a given degree.

Questions and exercises

E15. Indicate the main properties of a binomial tree. Draw a heap-ordered binomial tree B_3.

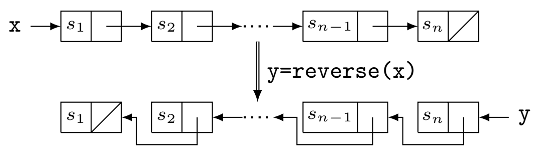
E16. Indicate some important properties derived from the definition of binomial heaps.

E17. Indicate some suitable ways to implement binomial trees and binomial heaps in `C++` .

E18. Consider simply-linked lists whose elements are instances of the class

```
struct Elem {
   string s;    // content
   Elem* next; // link to next element
};
```

Write an implementation of the function `Elem* reverse(Elem *x)`
which takes as input a pointer `x` to the first element of a list with n elements, and reverses destructively the list, by transforming

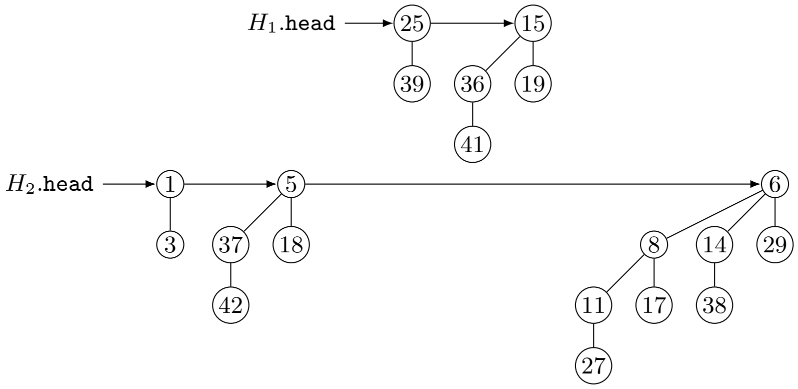


The implementation should have runtime complexity $O(n)$.

E19. Indicate the runtime complexities of the main operations on binomial heaps.

E20. Write down efficient implementations of the main operations on binomial heaps.

E21. Draw the binomial heap that results by merging the binomial heaps H_1 and H_2 depicted below:

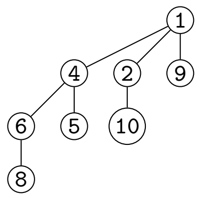


Answers

E15. A binomial tree B_k has the following properties:

1. It has 2^k nodes.
2. It has height k.
3. There are exactly $C(k, i)$ nodes at depth i for $i = 0, 1, \ldots, k$.
4. The root has degree k, which is greater than that of any other node. Moreover, if the children of the root are numbered from left to right by $k-1, k-2, \ldots, 0$, then child i is the root of a subtree B_i.

The following is a heap-ordered binomial tree B_3:



E16. Binomial heaps have the following important properties:

(a) The root of a heap-ordered tree contains the smallest key in the tree.

(b) A binomial heap with n nodes has at most $\lfloor \log_2 n \rfloor + 1$ binomial trees.

E17. Binomial trees can be implemented as instances of the `C++` class `BinTree`;

```
struct BinTree;
using BPtr = BinTree*;
struct BinTree {
   BPtr p;          // pointer to parent. It is NULL if this is a root node
   BPtr child;      // pointer to its left child
   BPtr sibling;    //  pointer to its right sibling
   int key;         // key
   int degree;      // number of children
};
```

A binomial heap for a set S of binomial trees can be conveniently implemented as the simply linked list of the trees of S in strictly increasing order of their degrees. We define the class

```
struct BHeap {
   BPtr head;
};
```

where `head` is a pointer to the root of the binomial tree from S with smallest degree. The roots of the binomial trees in S are linked through the `sibling` field of class `BinTree`.

E18. The following is a recursive implementation of `reverse()` with runtime complexity $O(n)$:

```
Elem* reverse(Elem* x) {
   if (x==NULL || x->next == NULL) return x;
   Elem* y = x->next;
   Elem* z = reverse(y);
   y->next = x;
   x->next = NULL;
   return z;
}
```

E19. The worst-case runtime complexities of the main operations on binomial heaps are:

makeHeap(): $O(1)$
insert(H, x): $O(\log n)$
minimum(H): $O(\log n)$
extractMin(H): $O(\log n)$
Union$(H1, H2)$: $O(\log n)$
decreaseKey(x, k): $O(\log n)$
del(H, x): $O(\log n)$

where n is the number of elements in the heap(s).

E20. The following are efficient implementations in `C++` of the main operations on binomial heaps. They operate on `C++` structures described in exercise E23, and make use of the auxiliary operations `binomialHeapMerge`, `binomialLink`, `nodeListMerge`, and `reverseList`:

```
BHeap* makeHeap() {
   BHeap* H = new BHeap();
   H->head = NULL;
   return H;
}
BPtr minimum(BHeap* H) {
   if (H == NULL || H->head==NULL) return NULL;
   BPtr m=H->head, crt=H->head;
   while (crt->sibling != NULL) {
       crt = crt->sibling;
       if (m->key>crt.key) m = crt;
   }
   return m;
}
BHeap* insert(BHeap *H, BPtr x) {
   BHeap* H1 = makeHeap();
   x->p = x->child = x->sibling = NULL;
   x->degree = 0;
   H1->head = x;
   BHeap* HH = Union(H,H1);
   delete H;
   delete H1;
   return HH;
}
BHeap* Union(BHeap* H1, BHeap* H2) {
   BHeap* H = makeHeap();
   H->head = binomialHeapMerge(H1,H2);
   if (H->head == NULL) return H;
   BPtr prev_x = NULL, x = H->head, next_x = x->sibling;
   while (next_x != NULL) {
      if (x->degree != next_x->degree ||
            (next_x->sibling && next_x->sibling->degree==x->degree)) {
         prev_x = x;                    // Cases 1 and 2
         x = next_x;                    // Cases 1 and 2
      } else if (x->key <= next_x->key) {
         x->sibling = next_x->sibling; // Case 3
         binomialLink(next_x, x);      // Case 3
      } else {
         if (prev_x == NULL)           // Case 4
            H->head = next_x;          // Case 4
         else
            prev_x->sibling = next_x;  // Case 4
         binomialLink(x, next_x);      // Case 4
         x = next_x;                   // Case 4
      }
      next_x = x->sibling;
   }
   return H;
}
```

```
void decreaseKey(BPtr x,int k) { ... }
void del(BHeap* H, BPtr x) {
   // Note: INT_MIN is the minimum value of an int in C++.
   // It is defined in <cstdint>
   decreaseKey(x,INT_MIN);
   extractMin(H);
}
BPtr extractMin(BHeap* H) {
   // find the root x with the minimum key from the root
   // list of H, and remove it from the root list of H
   BPtr x = minimum(H->head);
   if (x == NULL) { // H is empty
      return NULL;  // this heap is empty
   } else {
      BPtr n = H->head;
      // drop x from the root list of H
      if (n == x) H->head = n->sibling;
      else {
         while (n->sibling != x) n = n->sibling;
         n->sibling = n->sibling->sibling;
      }
   }
   BHeap* H1 = makeHeap();
   H1->head = reverseList(x->child);
   H->head = Union(H, H1)->head;
   x->child = x->sibling = x->p = NULL;
   return x;
}
BPtr nodeListMerge(BPtr l1, BPtr l2) {
   if (l1 == NULL) return l2;
   else if (l2 == NULL) return l1;
   else if (l1->degree <= l2->degree) {
      l1->sibling = nodeListMerge(l1->sibling, l2);
      return l1;
   } else {
      l2->sibling = nodeListMerge(l1, l2->sibling);
      return l2;
   }
}
BPtr reverseList(BPtr x) {
   if (x==NULL || x->sibling == NULL) return x;
   BPtr y = x->next;
   BPtr z = reverseList(y);
   y->next = x;
   x->next = NULL;
   return z;
}
BPtr binomialHeapMerge(BHeap* H1, BHeap* H2) {
   return nodeListMerge(H1->head,H2->head);
}
```

Computational geometry

This lecture was about:

- data structures to represent geometric objects (points, segments, vectors, polygons, etc.) in the Cartesian plane, and
- operations on such representations, and their geometric interpretation.

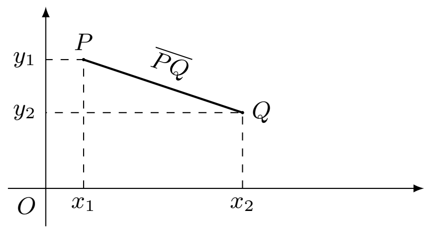
Points, segments, vectors, and operations on them

The simplest geometrical objects are points. A **point** P in plane is represented by its Cartesian coordinates: (x_P, y_P) where $x_P \in \mathbb{R}$ is the x-coordinate of P, and $y_P \in \mathbb{R}$ is the y-coordinate of P. For example, we can use `C++` to represent every point as an instance of the class

```
struct Point { float x, y; };
```

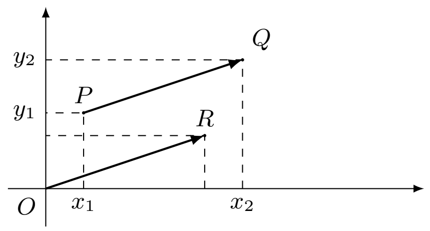
We write $P(x, y)$ to indicate that the Cartesian coordinates of a point P are (x, y). The point $O(0, 0)$ is the origin of the system of Cartesian coordinates.

The **segment** with endpoints P and Q is denoted by $\overline{PQ}$.



The **length** of $\overline{PQ}$ is $|PQ| = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$ if $P(x_1, y_1)$ and $Q(x_2, y_2)$. For example, if $P(1, 1)$ and $Q(4, 5)$ then $|PQ| = \sqrt{(4 - 1)^2 + (5 - 1)^2} = 5$.

The **vector** with origin $P(x_1, y_1)$ and destination $Q(x_2, y_2)$ is denoted by $\overrightarrow{PQ}$. The vector $\overrightarrow{PQ}$ has the same length and direction as the vector $\overrightarrow{OR}$ where $R(x_2 - x_1, y_2 - y_1)$. The vector $\overrightarrow{OR}$ is called the **shift** of $\overrightarrow{PQ}$ to the origin.



If two vectors $\overrightarrow{PQ}$ and $\overrightarrow{PR}$ have the same origin, we can compute their **cross product**: if $P(x_P, y_P)$, $Q(x_Q, y_Q)$, and $R(x_R, y_R)$ then $\overrightarrow{PQ}$ and $\overrightarrow{PR}$ are vectors with same origin P, and their cross product is

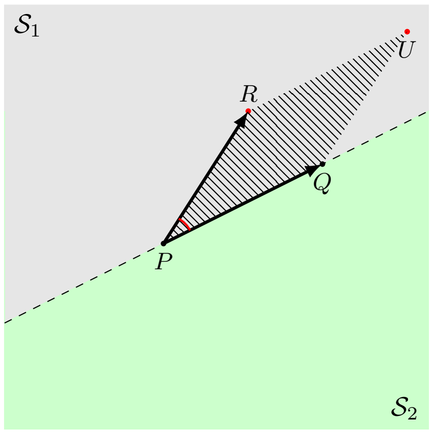
$$\overrightarrow{PQ} \times \overrightarrow{PR} = \det \begin{pmatrix} x_Q - x_P & x_R - x_P \\ y_Q - y_P & y_R - y_P \end{pmatrix} = (x_Q - x_P)(y_R - y_P) - (x_R - x_P)(y_Q - y_P).$$

Note that $\overrightarrow{PR} \times \overrightarrow{PQ} = \det \begin{pmatrix} x_R - x_P & x_Q - x_P \\ y_R - y_P & y_Q - y_P \end{pmatrix} = -\overrightarrow{PQ} \times \overrightarrow{PR}$.

The **geometric interpretation** of the cross product is derived from the formula

$$\overrightarrow{PQ} \times \overrightarrow{PR} = |PQ| \cdot |PR| \cdot \sin \widehat{QPR} \quad \text{where } \widehat{QPR} \text{ is measured counterclockwise around } P$$

and is easy to understand by looking at the picture below:



The line through P and Q splits the plane in two semiplanes: $\mathcal{S}_1$ (light gray) and $\mathcal{S}_2$ (light green). Observe that:

1. If $R \in \mathcal{S}_1$ then $\widehat{QPR}$ is an angle in the interval $(0°..180°)$, therefore $\sin \widehat{QPR} > 0$ and thus $\overrightarrow{PQ} \times \overrightarrow{PR} > 0$. In this case $\overrightarrow{PR}$ is turned counterclockwise with respect to $\overrightarrow{PQ}$.

2. If $R \in \mathcal{S}_2$ then $\widehat{QPR}$ is an angle in the interval $(180°..360°)$, therefore $\sin \widehat{QPR} < 0$ and thus $\overrightarrow{PQ} \times \overrightarrow{PR} < 0$. In this case $\overrightarrow{PR}$ is turned clockwise with respect to $\overrightarrow{PQ}$.

3. If R is on line PQ then $\widehat{QPR}$ is either $0°$ or $180°$, thus $\sin \widehat{QPR} = 0$ and $\overrightarrow{PQ} \times \overrightarrow{PR} = 0$.

Moreover, the absolute value $|\overrightarrow{PQ} \times \overrightarrow{PR}|$ of the cross product is the area of the parallelogram $QPRU$ between vectors $\overrightarrow{PQ}$ and $\overrightarrow{PR}$.

Consequences of the geometric interpretation of the cross product

1. The points R_1 and R_2 are on opposite sides of the line PQ iff the cross products $\overrightarrow{PQ} \times \overrightarrow{PR_1}$ and $\overrightarrow{PQ} \times \overrightarrow{PR_2}$ have different signs: $(\overrightarrow{PQ} \times \overrightarrow{PR_1}) \cdot (\overrightarrow{PQ} \times \overrightarrow{PR_2}) < 0$.

2. Three points P, Q, R are collinear iff $\overrightarrow{PQ} \times \overrightarrow{PR} = 0$.

3. The area of the triangle PQR is half of the area of the parallelogram between vectors $\overrightarrow{PQ}$ and $\overrightarrow{PR}$, which is $|\overrightarrow{PQ} \times \overrightarrow{PR}|/2$.

Exercises

In these exercises, we assume that all points are represented by instances of the class

`struct Point { float x,y; };`

and you can make use of the function

`float Cross(Point P, Point Q, Point R)`

which returns the value of the cross product $\overrightarrow{\mathtt{PQ}} \times \overrightarrow{\mathtt{PR}}$.

E22. Let `A`, `B`, `C` the vertices of a triangle, enumerated in clockwise order. Define the function

```
bool inside(Point P, Point A, Point B, Point C)
```

which returns `true` if point `P` is in the interior of triangle `ABC`, and `false` otherwise.

E23. Let `P1`, `P2` be the bottom left nodes, and Q_1, Q_2 the top right corners of two rectangles with edges parallel with the axes of coordinates. Define the functions

(a) `bool overlap(Node P1, Node P2, Node Q1, Node Q2)`
which yields `true` if the rectangles overlap, and `false` otherwise. The rectangles overlap if their intersection has a non-zero area.

(b) `float area(Node P1, Node P2, Node Q1, Node Q2)`
which computes the area of the overlap of these two rectangles. If the rectangles do not overlap, return 0.

E24. Suppose `v` is a vector of `n` ≥ 2 points. Define the function

```
float minArea(vector<Node> v)
```

which computes the area of the smallest rectangle which satisfies the following conditions: (1) its edges are parallel with the axes of coordinates; and (2) all points of `v` are in the rectangle.

E25. Write an implementation of the function

```
float dist(Point X,Point Y,Point Z)
```

which computes the distance from point `P` to line `YZ`.

E26. Write an implementation of the function

```
bool inside_circle(Point A, Point  B, Point C, Point O, float r)
```

which returns `true` if the circle with center `O` and adius `r` is inside triangle `ABC`. You can use the functions `bool inside()` from E26 and `dist()` from E29.

E27. Suppose `v` is a vector of points which contains the points $P_1, P_2, \ldots, P_n$ of a convex polygon in counterclockwise order.

(a) Write an implementation of the function

```
double area(vector<Point> v)
```

which computes the area of the polygon $P_1P_2 \ldots P_n$.

(b) Write an implementation of the function

```
double perim(vector<Point> v)
```

which computes the permieter of the polygon $P_1P_2 \ldots P_n$.

Answers

E22. `P` is in the interior of triangle `ABC` iff the following three conditions hold simultaneously:

1. `P` and `A` are on the same side of line `BC`
2. `P` and `B` are on the same side of line `AB`
3. `P` and `B` are on the same side of line `AC`

Two points `U` and `V` are on the same side of a line `XY` iff $(\overrightarrow{\mathtt{XY}} \times \overrightarrow{\mathtt{XU}}) \cdot (\overrightarrow{\mathtt{XY}} \times \overrightarrow{\mathtt{XV}}) > 0$.

From these observations we can derive an implementation of the function `inside()`.

Amortized Analysis

Amortized analysis is a technique used to estimate the time required to perform a sequence of operations, by taking the average over all the operations performed. It differs from average-case type analysis in the following ways:

- Average-case time analysis takes into account the probabilities of occurrences of different cases.
- Amortized-case analysis does not involve probabilities; it computes the average performance of each operation in the worst case. For example, amortized-case analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operation might be expensive.

The most common methods of amortized analysis are: the aggregate method, the accounting method, and the potential method.

Questions and exercises

E28. What is amortised analysis and how does it differ from average-case time analysis?

E29. Describe the aggregate method for amortized analysis.

E30. Describe the accounting method for amortized analysis.

E31. Describe the potential method for amortized analysis.

Fibonacci heaps

A Fibonacci heap is an advanced data structure designed to support well the following operations on dynamic sets (same as binomial heaps):

1. makeHeap(): creates and returns a new empty heap.
2. insert(H, x): inserts node referred by x, whose key field has already been filled in, into heap H.
3. minimum(H): returns a pointer to the node with minimum key in heap H.
4. extractMin(H): deletes the node with minimum key from heap H, and returns a pointer to the deleted node.
5. Union$(H1, H2)$: creates and returns a new heap that contains all the nodes of heaps $H1$ and $H2$. Heaps $H1$ and $H2$ are destroyed by this operation.
6. decreaseKey(x, k): assigns to node referred by x within the heap the new key value k. It is assumed that $k \leq x \rightarrow key$.
7. del(H, x): deletes the node referred by x from heap H.

Before defining Fibonacci heaps, we define unordered binomial trees. An **unordered binomial tree** is an element of the set of unordered trees $\{U_k \mid k \in \mathbb{N}\}$ defined recursively as follows:

U_0 consists of a single node.
For any $k > 0$, U_k is obtained from two trees U_{k-1}, for which the root of one is made into any child of the root of the other.

A rooted unordered tree is heap-ordered if, for every node x except the root, the key of x is larger than the key of the parent of x. A **Fibonacci heap** is a container for a set H of heap-ordered rooted unordered trees, where each node x contains

- the fields
    - ▶ x.`key` and possibly more fields for satellite data associated with key `key`.
    - ▶ x.`degree`: number of children in the child list of node x.
    - ▶ x.`mark`: a boolean value, which indicates whether node x has lost a child since the last time x was made the child of another node. Newly created nodes are unmarked.
- the pointers
    - ▶ x.`p`: pointer to the parent node. If x is a root node then x.`p` = `NULL`.
    - ▶ x.`child` pointer to any one of its children.
    - ▶ x.`left` and x.`right` which point to the left and right siblings of x, respectively. If x has no siblings. then x.`left` = x.`right` = pointer to x.

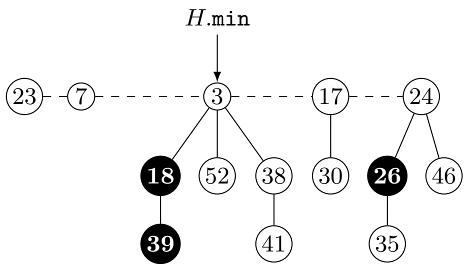
The children of a node x are linked together in a circular, doubly linked list, called the **child list** of x. The order in which children appear in the child list is arbitrary.

The root nodes of the trees in H are linked together into a circular, doubly linked list, using the `left` and `right` fields of the root nodes. This circular list is called the **root list** of the Fibonacci heap. The order of the trees in the root list is arbitrary.

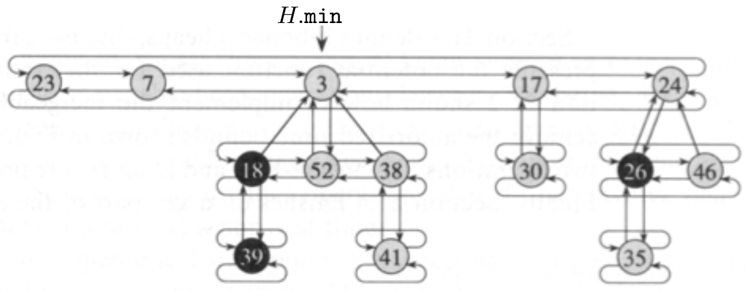
In addition, H has two fields:

- H.`min`: a pointer to the root of the tree in H with minimum key; this node is called the **minimum node** of the Fibonacci heap.
- H.`n`: the number of nodes currently in H.

For example, the figure below is a diagrammatic representation of a Fibonacci heap with five heap-ordered trees. The dashed line indicates the root list. The marked nodes are blackened (there are three marked nodes).



A more complete representation of the previous heap, showing the pointers `p` (up arrows), `child` (down arrows), `left` and `right` (sideways arrows), is illustrated below.

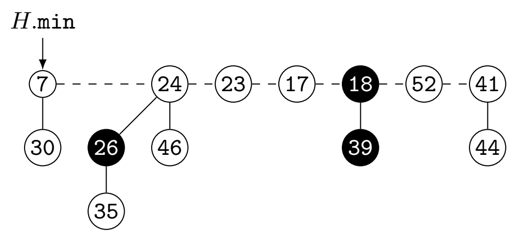


Fibonacci heaps perform better than binomial heaps because the amortized cost of their operations is better.

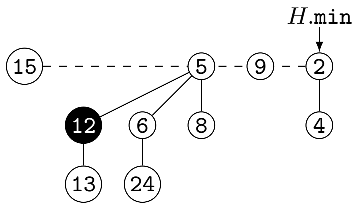
Questions and exercises

E32. What are the amortized costs of the main operations on Fibonacci heaps?

E33. An important auxiliary operation on Fibonacci heaps is the consolidation of the root list. Draw the Fibonacci heap produced by consolidating the root list of the Fibonacci heap



E34. Draw the Fibonacci heap produced by consolidating the root list of the Fibonacci heap



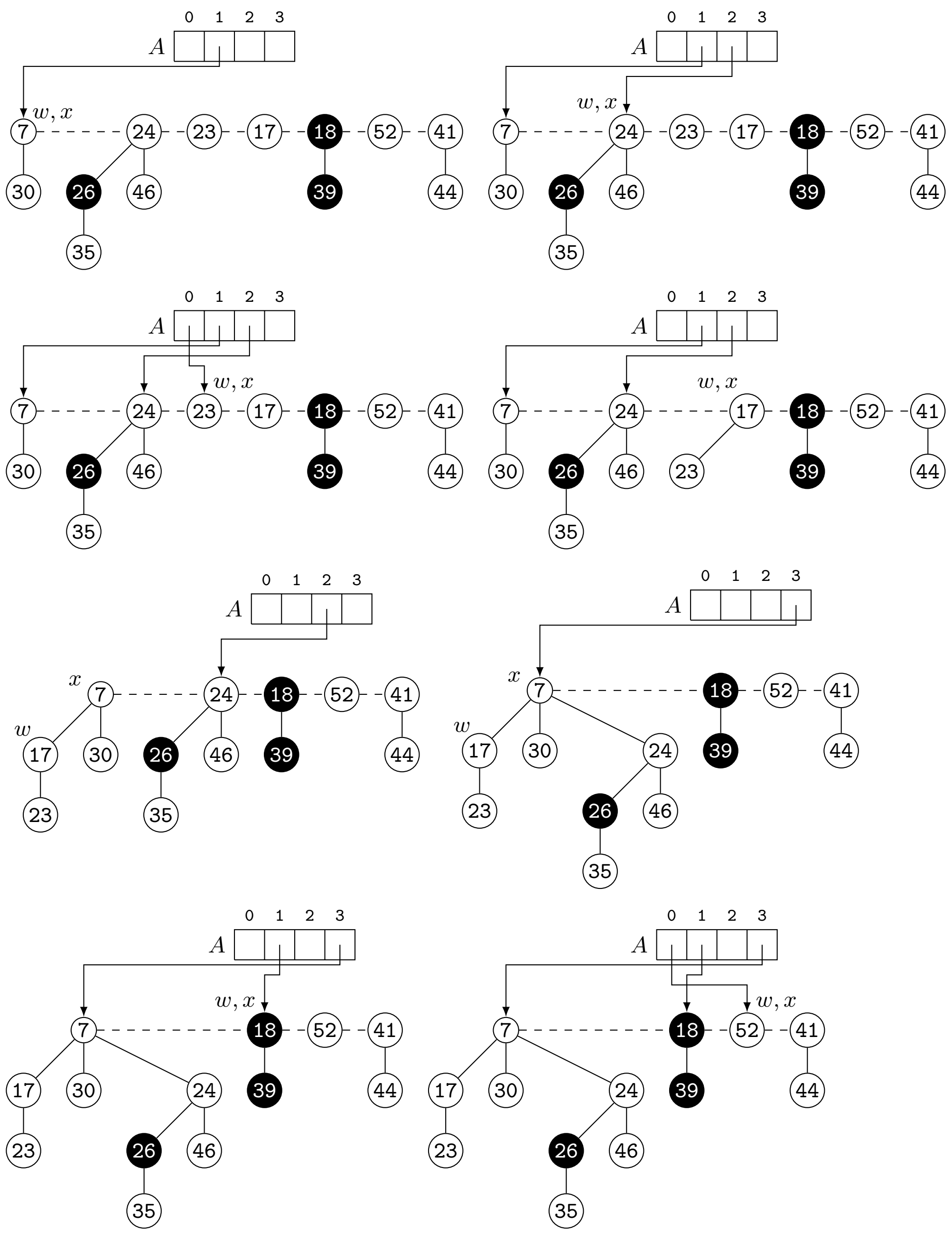
Answers

E32. The amortized cost of the main operations on binomial heaps are:

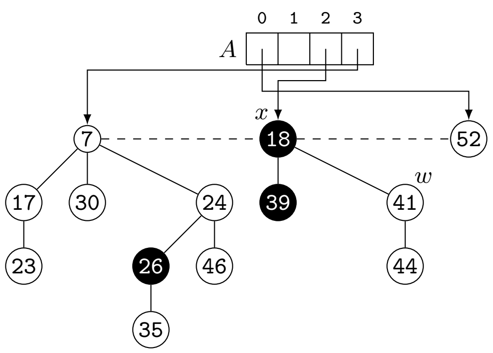
makeHeap(): $O(1)$
insert(H, x): $O(1)$
minimum(H): $O(1)$
extractMin(H): $O(\log n)$
Union$(H1, H2)$: $O(1)$
decreaseKey(x, k): $O(1)$
del(H, x): $O(\log n)$

where n is the number of elements in the heap(s).

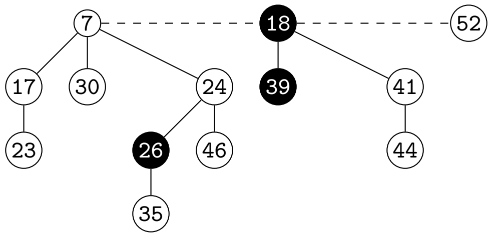
E33. The number of nodes in H is 13, and an upper bound on the maximum degree of any node in Fibonacci heaps with 13 nodes is $D(13) = \log_2 13 < 4$. This means that, during consolidation we will encounter only trees with degrees between 0 and 3. Therefore, for the consolidation of H we allocate an array A of size 4 (for pointers to trees of degree 0, 1, 2, or 3). The consolidation algorithm traverses from left to right the root list of H, starting from $w = H$.min until $w =$ the left sibling of H.min (which is the node with key 41). The consolidations performed during this traversal are shown in the picture below:



A
0 1 2 3
w, x
x
w
7 24 23 17 18 52 41
30 26 46 39 44
35

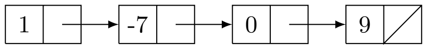


Thus, the result of consolidation is the Fibonacci heap

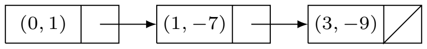


Data structures for symbolic computation

In this lecture, we described (1) the representation of polynomials by dense lists, and by sparse lists, and (2) some of the main operations that we wish to perform with polynomials. For example, the polynomial $f = 1 - 7\,x + 9\,x^3$ has the dense list representation



and the sparse list representation



Questions and exercises

E35. Illustrate the dense representation of the polynomial $f = 3\,x - 91\,x^4 + x^5 - x^6 + 4\,x^8$.

E36. Illustrate the sparse representation of the polynomial $g = 2916 + 3\,x^2 - x^{12} + x^{24} - 2\,x^{216}$.

E37. Consider polynomials represented by dense lists for the coefficients of its terms in increasing order of their degree, where the elements of the list are instances of the `C++` class

```
struct Term {
   float coeff;
   Term* next;
   Term(float c, Term* nxt) : coeff(c), next(nxt) { }
};
```

Define the following operations with polynomials:

1. `Term* pdiff(Term* f, Term* g)`
   which takes as inputs two pointers to the dense list representations of some polynomials, and returns a pointer to the dense list representation of their difference.
2. `Term* cprod(float c,Term* f)`
   which takes as inputs a number c and a pointer to the dense list representation of a polynomial f, and returns a pointer to the dense list representation of the polynomial $c \cdot f$.
3. `Term* x2quot(Term* f)`
   which takes as input the dense list representation of a polynomial f and returns the dense list representation of quotient of dividing f by $2\,x$.

Answers

E37. Operations with polynomials.

1.
```
Term* pdiff(Term* f,Term* g) {
   if (f==Nil) return minus(g);
   if (g==Nil) return f;
   Term* h = pdiff(f->next,g->next);
   float c = f->coeff - g->coeff;
   if (c==0 && h==Nil) return Nil;
   return new Term(c,h);
}

Term* minus(Term* g) {
  if (g==Nil) return Nil;
  else return new Term(-(g->coeff),minus(g->next));
}
```
2. ...
3.
```
// quotient of f(x)/(2*x)
Term* x2quot(Term* f) {
   if (f==Nil || f->next==Nil) // f is a constant polynomial
      return Nil;
   return cprod(f->next,1/2);
}

Term cprod(Term* g, float c) {
   return (g==Nil)?Nil:(new Term(c*g->coeff,cprod(g->next,c)));
}
```

Data structures for operations on strings

Strings are arrays of characters from a finite alphabet Σ. The most important operations with strings are:

1. String matching: finding all occurrences of a string $P[1..m]$ in another string $T[1..n]$.
   - P is the string we are looking for; we call it **pattern**
   - T is the string in which we search; we call it **text**.

   An occurrence of P in T is indicated by an index i in T such that $T[i..i+m-1] = P[1..m]$.
2. Finding all occurrences of a set of patterns $\{P_1, \ldots, P_z\}$ in a string.

The following data structures are used to implement efficiently the string operations:

- Finite automata.
- Keyword trees with failure links.
- Suffix trees with suffix links.
- Generalized suffix trees with suffix links.

Finite automata

are used when we search often the same pattern $P[1..m]$ in many texts. In such situations, we proceed as follows:

1. We compute a finite automaton $\mathcal{A} = (Q, q_0, A, \Sigma, \delta)$ for the pattern P. The construction of $\mathcal{A}$ for a pattern $P[1..m]$ can be done in $O(m)$ time.
2. We find all occurrences of P in T by reading text T with automaton $\mathcal{A}$.
   - A reads $T[1..n]$ in n steps $= O(n)$ time.
   - Every occurrence of P in T is detected when P enters a final state $q \in A$.

Remark: In the lecture notes, we indicated a method to construct $\mathcal{A}$ in $O(m^3)$ time, and mentioned that there exists a better method which constructs $\mathcal{A}$ in $O(m)$ time.

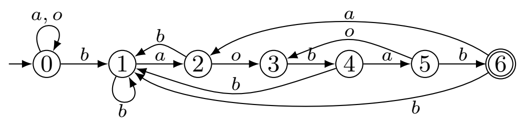
lllustrated example: Suppose $P[1..6] = baobab$. In general, the automaton for a string of length m has $m+1$ states: $Q = \{0, 1, \ldots, m\}$. Th intial state is $q_0 = 0$ and the final state id $m \in A$. For short patterns, like P, we compute $\mathcal{A}$ as follows:

1. We draw the states in increasing order, from left to right, and connect consecutive states with arrows labeled with the characters of P:

$$\begin{tikzcd} {} \arrow[r] & 0 \arrow[r, "b"] & 1 \arrow[r, "a"] & 2 \arrow[r, "o"] & 3 \arrow[r, "b"] & 4 \arrow[r, "a"] & 5 \arrow[r, "b"] & 6 \end{tikzcd}$$

2. The letters in P are from alphabet $\Sigma = \{b, a, o\}$. We must complete the state transition diagram such that from every state there are 3 outgoing transitions: a transition labeled with b, another transition labeled with a, and another transition labeled with o:

- We add $i \xrightarrow{u} j$ iff j is the largest number such that $P[1..j]$ is suffix of $P[1..i]u$.



The transitions that were not drawn go to state 0.

Keyword trees with failure links

are used when we search often the same patterns from a set $\{P_1, \ldots, P_z\}$ in many texts. In such situations, we use the Aho-Corasick algorithm, which proceeds as follows:

1. For the set of patterns $\mathcal{P} = \{P_1, \ldots, P_z\}$ we construct a tree-like data structure $\mathcal{K}$ called **keyword tree with failure links.** This data structure can be constructed in time $O(m)$ where m is the sum of lengths of patterns $P_1, \ldots, P_z$.
2. The occurrences of pattems from $\mathcal{P}$ in a text $T[1..n]$ are detected by traversing simultaneously
   (a) the tree $\mathcal{K}$: downward from root, and following the failure links when going down is impossible
   (b) the text $T[1..m]$ from left to right.

Suffix trees with suffix links

A suffix tree (with suffix links) is a tree-like data structure that can be constructed in time $O(n)$ for a large text $T[1..n]$. It can be used to find all occurrences of a pattern $P[1..n]$ in T in time $T(m + k)$ where k is the number of occurrences of P in T.

Questions and exercises

E38. Draw the transition diagram of the matching automaton for the pattern $P = salsa$.

E39. Draw the keyword tree with failure links for the set of patterns $\{GTA, AGT, AAC\}$

E40. Draw the suffix tree with suffix links for the text `mississippi`

Binary heaps

E41. What is a binomial heap? Indicate the main operations supported by binary heaps, and their runtime complexity.

E42. Write down the pseudocode of the algorithm for heap-sort, using a binomial heap.

kd-trees

E43. Draw a balanced kd-tree for the set of points

$$\{(1,3,1),(2,1,6),(3,6,5),(4,4,2),(5,10,3),(6,5,8),(7,7,12)\}$$

Sorting in sublinear time

E44. Write down the pseudocode of counting sort.

E45. How many assignments are performed by counting sort, when it is used to sort an array of 20 elements with values between 0 and 6?

A Binary search trees

maximum

```
// finds the node with maximum key in tree with root r
Node* tree_maximum(Node* r) {
   if (r == Nil) return r;
   while (r->right != Nil) r=r->right;
   return r;
}

// finds the node with minimum key in tree with root r
Node* tree_minimum(Node* r) {
   if (r == Nil) return r;
   while (r->left != Nil) r=r->left;
   return r;
}

Node* predecessor(Node* x) {
   if (x == Nil) return x;
   if (x->left!=Nil) return tree_maximum(x->left);
   Node* y = x->p;
   while (y!=Nil && x==y->left) {
      x=y;
      y=y->p;
   }
   return y;
}

Node* successor(Node* x) {
   if (x == Nil) return x;
   if (x->right!=Nil) return tree_minimum(x->right);
   Node* y = x->p;
   while (y!=Nil && x==y->right) {
      x=y;
      y=y->p;
   }
   return y;
}
```

```
// finds the node with maximum key in the
// tree which contains this node
Node* maximum(Node* x) {
    Node* z = x;
    Node* y = x->p;
    while (y!=Nil && z==y->right) {
        z=y;
        y=y->p;
    }
    if (y==Nil) return tree_maximum(x);
    z = y->p;
    while (z!=Nil && y==z->left) {
        y=z;
        z=z->p;
    }
    if (z==Nil) return y;
    return tree_maximum(z);

}
```

B Red-Black trees

RBInsertFixup

```
RBInsertFixup(Node* $z$) // method of class RBNode
 1 while $z$->p->color == RED
 2       if $z$->p == $z$->p->p->left
 3       then $y$ = $z$->p->p->right
 4            if $y$->color == RED
 5              then $z$->p->color = BLACK            // Case 1
 6                   $y$->color = BLACK               // Case 1
 7                   $z$->p->p->color = RED           // Case 1
 8                   $z$ = $z$->p->p                  // Case 1
 9              else if $z$ == $z$->p->right
10                     then $z$ = $z$->p              // Case 2
11                          LeftRotate($z$)           // Case 2
12                   $z$->p->color = BLACK            // Case 3
13                   $z$->p->p->color = RED           // Case 3
14                   RightRotate($z$->p->p)           // Case 3
15       else
16            (same as then clause with right and left exchanged)
17 root->color = BLACK
```

RBDeleteFixup

```
RBDeleteFixup($x$)
while $x \neq$ root and $x$->color == Black
    if $x$ == $x$->p->left
        $w$ = $x$->p->right
        if $w$->color = Red
            $w$->color = Black                                      Case 1
            $x$->p->color = Red                                     Case 1
            LeftRotate($x$->p)                                      Case 1
            $w$ = $x$->p->right                                     Case 1
        if $w$->left->color == Black and $w$->right->color == Black
            $w$->color = Red                                        Case 2
            $x$ = $x$->p                                            Case 2
        else if $w$->right->color == Black
                $w$->left->color = Black                            Case 3
                $w$->color = Red                                    Case 3
                RightRotate($w$)                                    Case 3
                $w$ = $x$->p->right                                 Case 3
            $w$->color = $x$->p->color                              Case 4
            $x$->p->color = Black                                   Case 4
            $w$->right->color = Black                               Case 4
            LeftRotate($x$->p)                                      Case 4
            $x$ = root                                              Case 4
    else (same as then clause, with right and left exchanged)
$x$->color = Black
```