Topic 10

Heap, Set and Map

資料結構與程式設計
Data Structure and Programming

12/05/2018

Linear Data Types

- In previous topic and Homework #5, we have learned linear data types like list and array
  - Tradeoffs between insert/delete/find operators
  - Memory overhead

  ➔ Constant time for "push_back()" or "push_front()" operation

- The best way to use linear data types is ---
  - Data are recorded in a linear sequence (i.e. only push_back or push_front is needed)
  - Linearly traverse each element (i.e. for(...; li++))
  - No "find", "insert any", nor "delete any"

    ➔ If needed, use "tree"?

Consider the Scenario...

- Suppose we are assigning jobs sequentially to several machines ---
  - One job to one machine and we record the accumulated runtime for each machine.
  - Our machine selection criteria is to "even out" the runtime of the machines.
  - In other words, we would like to pick the machine with least accumulated runtime for the next job

  ➔ Do we need to sort ALL the elements?

  ➔ Need a priority queue

Priority Queue

- An ADT that supports 2 operations
  - Insert
  - Delete min(or max)
- An element with arbitrary priority can be inserted to the queue
- At any time, it should take constant time to find the element with min(or max) priority and could efficiently remove it from the list
  - Need to figure out which is the one with next lowest(highest) priority efficiently

Using List or Array?

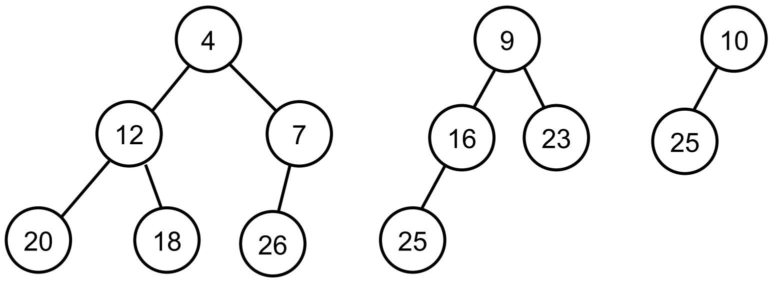
- Use linear ADT with an extra field to record the element with min(max) priority
  - Insert: O(1)
  - Delete min(max): O(n)

  (why?)

- As we learn before, O(n) is not good. We would prefer an ADT with O(log n) for both operations

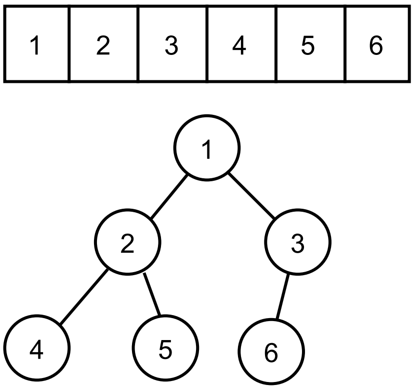
Min (Max) Heap

- A complete binary tree in which the key value in each node is no larger (smaller) than its children



Why complete binary tree?

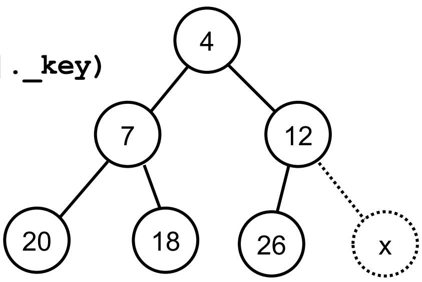
Remember that we can use array to implement a complete binary tree...



- Parent
  = child / 2
- Child
  = Parent * 2
  or Parent * 2 + 1

MinHeap Insertion

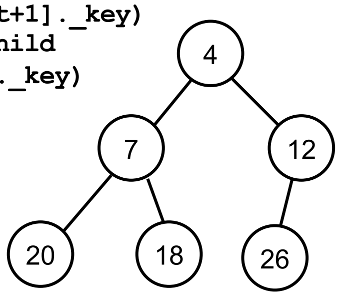
```
// Let n be the index of the last element
void MinHeap::insert(const T& x)
{
  int t = ++n; // next to the last
  while (t > 1) {
    int p = t / 2;
    if (x._key >= _heap[p]._key)
      break;
    _heap[t] = _heap[p];
    t = p;
  }
  _heap[t] = x;
}
```



What's the time complexity?

Delete Min Element

```
T& MinHeap::deleteMin()
{
  T ret = _heap[1];
  int p = 1, t = 2 * p;
  while (t <= n) {
    if (t < n) // has right child
      if (_heap[t]._key > _heap[t+1]._key)
        ++t; // to the smaller child
    if (_heap[n]._key < _heap[t]._key)
      break;
    _heap[p] = _heap[t];
    p = t;
    t = 2 * p;
  }
  _heap[p] = _heap[n--];
  return ret;
}
```



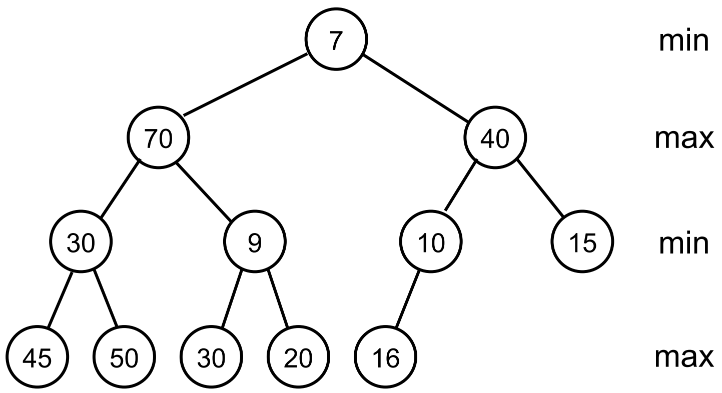
What's the time complexity?

Min(Max) Heap

- Simple implementation (just an array)
- Good insertion and deleteMin complexity
  - O(log n) vs. O(n)

What if you want to
delete min AND delete max?

Min-Max Heap



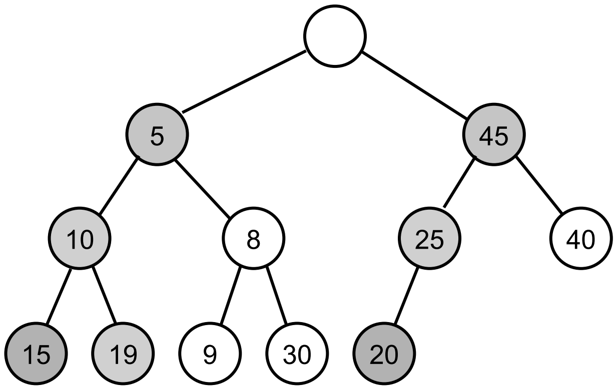
- Insert, delete min, delete max: all O(log n) (why?)

Deap

◆ Double-ended heap

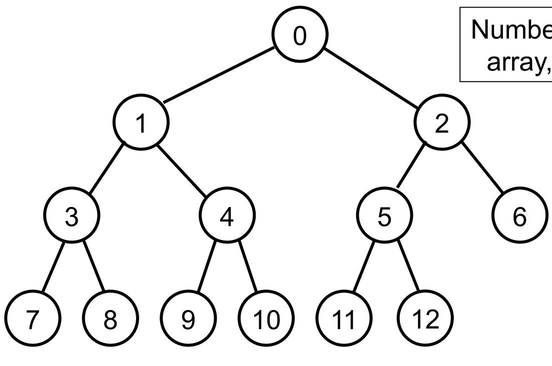
1. The root contains no element
2. The left subtree is a min heap
3. The right subtree is a max heap
4. Let i be any node in the left subtree. Let j be the corresponding node in the right subtree. If such a j node does not exist, then let j be the corresponding parent of i.
   ➔ The key in node i is less than or equal to that in j.

Deap Example



- Insert, delete min, delete max: all O(log n) (why?)
  - But faster than min-max heap by a constant factor
  - Algorithm is simpler

Deap Implementation



Numbers are indices in array, not data value

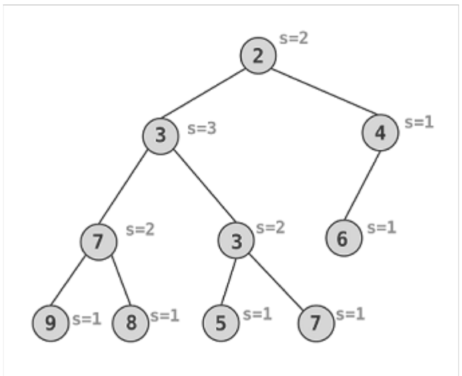
- Given a node 'i', how to find the "corresponding parent" or "corresponding child"?
- When insertion or deletion, what should we do when the node value is greater/smaller than its corresponding parent/child?

Practice #1

- Write the pseudo codes for the "insert", "delete min", and "delete max" operations of the min-max heap.
- Write the pseudo codes for the "insert", "delete min", and "delete max" operations of the deap.

More Varieties of Heaps: Leftist Heap

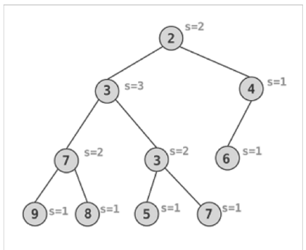
- In contrast to a *binary heap*, a leftist heap attempts to be very unbalanced.
  - s-value(v): the distance to the nearest leaf.
  - In addition to the heap property, the right child of each node has the lower s-value.



- Support "combine(heap1, heap2)" in O(log n)

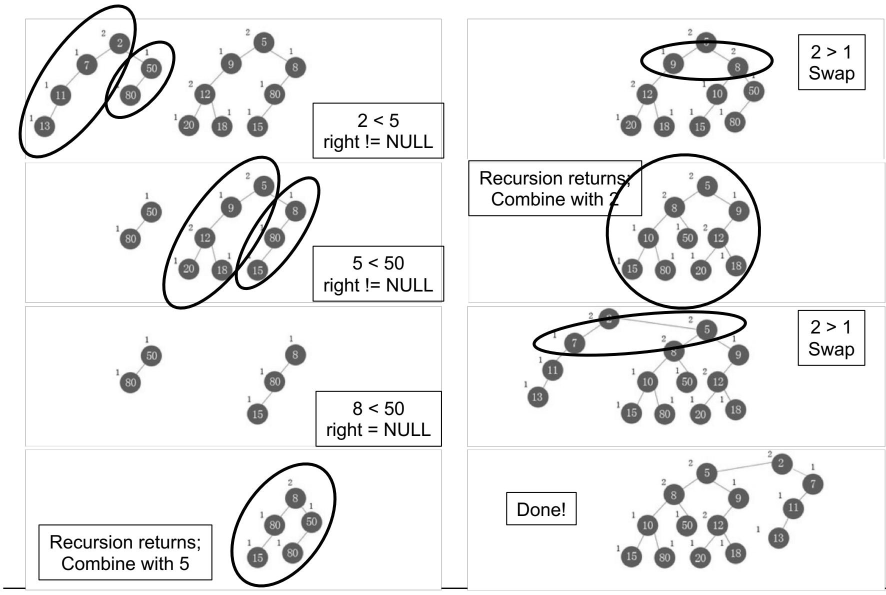
Leftist Heap: Huh?

- Remember: “combine(heap1, heap2)” in O(log n)
  - Both“insert” and “deleteMin” operations can be realized by “combine”. (How?)




```
combine(h1, h2) {
  compare(min(h1), min(h2));
  // let min(hi) < min(hj)
  if (right(hi) == NULL)
    right(hi) = hj;
  else
    combine(right(hi), hj);
  // hj is now the combined heap
  if (s(right(hj)) > s(left(hj))
    swap(right(hj), left(hj));
}
```


Leftist Heap: Combine [src] http://blog.yam.com/rockmanray/article/44962825

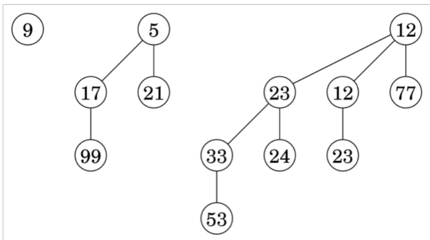


Leftist Heap: Discussion

- How can “insert” and “delete” be implemented by “combine”?
- Can we implement “combine” efficiently in a min-heap?
- Can the leftist heap be implemented in an array, like a min-heap does?

More Varieties of Heaps: Binomial heap

- Binomial tree of order k
  - Binomial tree of order 0 is a single node
  - The root of a binomial tree of order k has k children, who are roots of binomial trees of order k-1, k-2,..., 0
  - Has exactly 2^k nodes; height = k
- Binomial heap
  - A collection of Binomial trees
  - Most operations have the complexity O(log n)
  - But the amortized complexity is either O(1) or O(log n)



Binomial Heap: Properties

- Given a binomial heap with n nodes:
  - The node containing the min element is a root of $B_0, B_1, \ldots,$ or B_k.
  - It contains the binomial tree B_i iff $b_i = 1$, where $b_k \ldots b_2\, b_1\, b_0$ is binary representation of n.
  - It has $\leq \lfloor \log_2 n \rfloor + 1$ binomial trees.
  - Its height $\leq \lfloor \log_2 n \rfloor$.

[src] http://www.cs.princeton.edu/~wayne/kleinberg-tardos/pdf/BinomialHeaps.pdf

Binomial Heap: Operations

- Similar to Leftist Heap, the operations of Binomial Heap can be realized by the “compose” (aka. “meld”) operation.
- Compose operation:
  - Binary addition
  - Given two binomial heaps
    $H_1 := \{ (B_3, B_2, B_1, B_0) = (1, 1, 0, 1) \}$, and
    $H_2 := \{ (B_4, B_3, B_2, B_1, B_0) = (1, 0, 1, 0, 1) \}$.
    The composed binomial heap
    $H_m := \{ (B_5, B_4, B_3, B_2, B_1, B_0) = (1, 0, 0, 0, 1, 0) \}$.

Binomial Heap: Compose Operation

- Atomic operation:
  - Given two binomial trees B_i, B_j, with the *same order* k, then compose(B_i, B_j):
  1. Connect the roots r_i, r_j of B_i, B_j.
  2. Choose min(r_i, r_j) as the root of the composed tree
  3. The composed tree is of order k+1

  ➔ What's the time complexity? O(1)

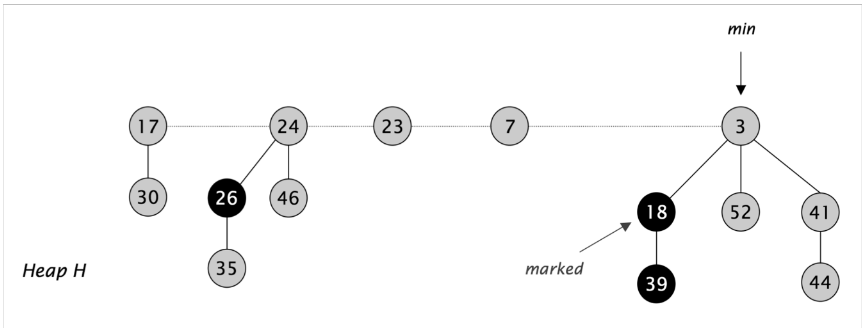
  ➔ What if we have three binomial trees with the same order?
- The compose operation of two binomial heaps:
  1. Align the binomial trees of both heaps
  2. From the trees with the least order, perform tree composition
  3. Propagate to the next order of tree if necessary
- What's the time complexity? O(log n)

Binomial Heap: Other Operations

- FindMin
  // remember: It has ≤ $\lfloor \log_2 n \rfloor$ + 1 binomial trees
  - O(log n)
- DeleteMin
  - Note: after the "min" is removed, the corresponding binomial tree (of order k) is broken and becomes k binomial trees
  - It just becomes "compose" operations of some binomial trees // How many? Compose of 2 or 3?
  - O(log n)
- DeleteNode(iterator pos)
  - O(log n)
- Insert(x)
  - O(log n)

More Varieties of Heaps: Fibonacci heap

- Fibonacci heap
  - Especially useful when deleteMin() & delete(n) are rarely called ➔ amortized O(log n)
  - All other operations are O(1)



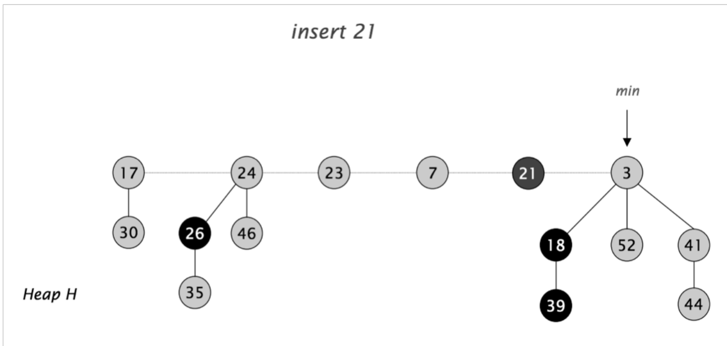
Fibonacci Heap

- Basic idea
  - Similar to binomial heaps, but less rigid structure
  - Binomial heap: eagerly consolidate trees after each insert (maintain binomial structure)
  - Fibonacci heap: lazily defer consolidation until next **<u>delete-min</u>**
- Properties
  - Set of heap-ordered trees.
  - Maintain pointer to minimum element
  - Set of marked nodes (if one of its children is removed)

(Ref) https://www.cs.princeton.edu/~wayne/teaching/fibonacci-heap.pdf

Fibonacci Heap: Insert Operation

- Create a new singleton tree.
- Add to root list; update min pointer (if necessary) ➔ O(1)



(Ref) https://www.cs.princeton.edu/~wayne/teaching/fibonacci-heap.pdf

Fibonacci Heap: DeleteMin Operation

- Let H be a Fibonacci heap and x be a node
  - rank(x): number of children of node x
  - rank(H): max rank of any node in heap H
  - tree(H): number of trees in heap H
- DeleteMin
  - Delete min; meld its children into root list; update min
  - Consolidate trees so that no two roots have same rank // What about the min pointer?

  ➔ Time complexity: O(rank(H)) + O(trees(H))

  ➔ Amortized cost: O(rank(H))

Heap Operations Supported in STL

- STL does not have a “heap” class
  - Instead, it support several operations that can operate on “array” like data structure
- Operations
  - void make_heap(first, last[, comp]);
  - void push_heap(first, last[, comp]);
  - void pop_heap(first, last[, comp]);
  - void sort_heap(first, last[, comp]);
  - bool is_heap(first, last[, comp]);

  ➔ *fist, last: RandomAccessIterator*

  ➔ *comp: StrictWeakOrdering (optional)*

Heap Operations Supported in STL (Example)

```
{
   int size = 100;
   int arr[size];
   // write data to arr
   …
   make_heap(arr, arr+size);
   int min = arr[0];
   pop_heap(arr, arr+(--size));
   arr[size++] = min*min;
   push_heap(arr, arr+size);
   sort_heap(arr, arr+size);
   // heap is now sorted
}
```

Summary: Heap Structures

- Pros:
  1. Good complexity of “insert”, “delete min(max)”, ... operations
  2. Simple data structure (low memory overhead)
  3. Simpler algorithms (than BST)

- Con
  1. Data are not sorted
     ➔ Still have O(n) for “find” operation

Review: Binary Search Trees

- Binary Search Trees (BSTs)
  - Left subtree ≤ this ≤ right subtree
  - Complexity depends on the height of the tree
  - Worst case: can be degenerated as a tree with height O(n)
- Balanced BSTs
  - The heights of left subtree and right subtree are somewhat balanced
    - Height ~ O(log n)
  - Examples: AVL, 2-3, 2-3-4, red-black, splay trees
  - Algorithms for their operations are complicated

Sorted ADT in STL

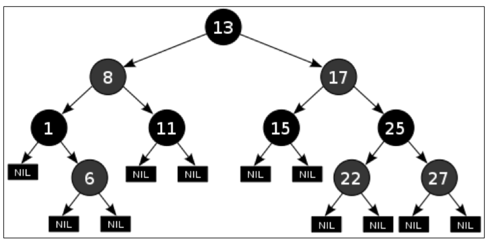
- Also classified as "Associative Containers"

1. set
2. multiset
3. map
4. multimap

➔ Implemented in "red black tree"

Red Black Tree (More to cover later)

- A node is either red or **black**. The root is **black**
- All leaves are black (i.e. All leaves are same color as the root.)
- Every red node must have two **black** child nodes.
- Every <u>path</u> from a given node to any of its descendant leaves contains the same number of **black** nodes.
- Memory efficient
- Although balancing is NOT perfect, O(log n) for insert, delete, and find



class set in STL

- To store elements in a set
  - e.g. { 2, 3, 5, 7, 9 }
- set<Key[, Compare, Alloc]>
  - class Key: element type
  - class Compare: how the elements are compared (optional; default = less<Key>)
  - class Alloc: used for internal memory management (optional; default = alloc)

Member Functions in class set

1. iterator begin() const;
   iterator end() const;
2. pair<iterator, bool> insert(const value_type& x);
   iterator insert(iterator pos, const value_type& x);
   void insert(InputIterator, InputIterator);
3. void erase(iterator pos);
   size_type erase(const key_type& k);
   void erase(iterator first, iterator last);
4. iterator find(const key_type& k) const;
5. size_type count(const key_type& k) const;
6. iterator lower_bound(const key_type& k) const;
   iterator upper_bound(const key_type& k) const;
   pair<iterator, iterator> equal_range(const key_type& k) const;

Other Functions for class set

1. includes
   - Check if one set is included in another
2. set_union
3. set_intersection
4. set_difference
5. set_symmetric_difference
   - (A – B) U (B – A)

class multiset in STL

- Unlike "set", where elements with same value are stored only once, in multiset, they can be stored repeatedly
  - e.g. { 2, 3, 5, 5, 6, 7, 7, 7 }
- multiset<Key[, Compare, Alloc]>
  - class Key: element type
  - class Compare: how the elements are compared (optional; default = less<Key>)
  - class Alloc: used for internal memory management (optional; default = alloc)

class map in STL

- In many applications, data are associated with keys (or id's)
  - For example, (id, student record)
  - e.g. { (Mary, 90), (John, 85), (Sam, 71) ... }
- class map<Key, Data[, Compare, Alloc]>
  - class Key: compared data type
  - class Data: value type
  - class Compare: how the elements are compared (optional; default = less<Key>)
  - class Alloc: used for internal memory management (optional; default = alloc)

Example of using class map (1)

```
map<string, unsigned>  scoreMap;
scoreMap["Mary"] = 90;
scoreMap["John"] = 85;
scoreMap["Sam"] = 71;
unsigned maryScore = scoreMap["Mary"];
cout << "Mary's score = " << maryScore << endl;
map<string, unsigned>::iterator mi;
mi = scoreMap.find("John");
if (mi != scoreMap.end())
  cout << "John's score = " << (*mi).second << endl;
```

➔ How about "map<const char*, unsigned>"?

Comments about map::operator []

- Since operator[] might insert a new element into the map, it can't possibly be a const member function.
- Note that the definition of operator[] is somehow tricky: m[k] is equivalent to (*((m.insert(value_type(k,data_type()))).first)).second.
  - value_type = pair<Key, Data>
  - insert(value_type) returns a pair<map::iterator, bool>
- Strictly speaking, this member function is unnecessary: it exists only for convenience.

http://www.sgi.com/tech/stl/Map.html

Bad example of using class map

```
map<const char*, unsigned> mmm;
map<const char*, unsigned>::iterator mi;
char buf[1024];
cin >> buf; mmm[buf] = 10;
cin >> buf; mmm[buf] = 20;
cin >> buf; unsigned s1 = mmm[buf];
cout << buf << " = " << s1 << endl;
cin >> buf; unsigned s2 = mmm[buf];
cout << buf << " = " << s2 << endl;
```

Example of using class map (2)

```
string str;
for (int i = 0; i < 5; ++i) {
  cin >> str; mm.insert(pair<string, int>(str, i));
}
while (1) {
  cin >> str;
  map<string, int>::iterator mi = mm.find(str);
  if (mi == mm.end()) {
    cout << "Not found!!" << endl;
    break;
  }
  cout << (*mi).first << " = " << (*mi).second << endl;
}
```


Conclusion: Set and Map

- "set" and "map" are useful data structures when we need to perform efficient "insert", "erase", and "find" operations
  - Usually implemented by balanced binary search trees
  - Implementation efforts can be high
  - Using STL may be a good choice
- Remember, unbalanced BSTs may not be a bad choice
  - Most randomly inserted BSTs are somewhat balanced
- Remember, there's no free lunch
  - Overhead in insert (vs. push_back)
  - If we don't need to do "erase" or "find" during insertions... (what's the alternative?)