Intro to Trees, and Priority Queues/Heaps

Informal Definition: Tree

Mathematical abstraction that plays a central role in the design and analysis of algorithms

- build and use explicit data structures that are concrete realizations of trees
- describe the dynamic properties of algorithms

Formal Definition: Tree

- Tree: nonempty collection of vertices/nodes and edges in which there exists precisely one path connecting any two nodes

(Usually, we say graphs have vertices, and trees have nodes)
Some Tree Terminology

…In the context of FindFib(5)
  - Root: “top-most” vertex in the tree
    - the initial call
    - e.g., FindFib(5)
  - Parent/Child: direct links in tree
    - FindFib(5) calls FindFib(4) then FindFib(3)
      (parent) \( (1^{st} \text{ child}) \) \( (2^{nd} \text{ child}) \)
  - Internal node: a node with children
    - Any call to FindFib with argument \( \geq 2 \)
      - e.g., FindFib(3), FindFib(2)
  - Leaf/External node: a node without children
    - Any call to FindFib(0) or FindFib(1)

A Special Case: Binary Tree

Def’n 5.1: A binary tree is either a leaf node or an internal node connected to a pair of binary trees, which are called the left subtree and the right subtree of that node

- M-ary tree: each internal node has exactly \( M \) children.

Complete Binary Tree Property

- We want the heap to be as small in height as possible.
- Def’n: complete binary tree
  - A heap with height \( h \) where
    - levels 0, 1, 2, …, \( h-1 \) have the max number of nodes possible
    - all internal nodes are to the left of the external nodes in level \( h-1 \)
Concrete Implementation

Node in Binary Tree

```c
struct node {
    Item item;
    node *left, *right;
};
```

- A node contains some information, and points to its left child node and right child node
- Efficient for moving down a tree from parent to child
- Modification to move up tree from child to parent?

Priority Queue: ADT

- Efficient insertion of new items
- Efficient removal of item with largest key

Definition 9.1: A priority queue is a data structure of items with keys that supports two basic operations: insert a new item, and remove the item with the largest key

Comparing Keys:

- A key can be an arbitrary object
- A priority queue needs a comparison rule which defines a total order relation:
  - Reflexive: K ≤ K
  - Antisymmetric: if K1 ≤ K2 and K2 ≤ K1, then K1=K2
  - Transitive: if K1 ≤ K2 and K2 ≤ K3, then K1 ≤ K3
- Such a rule defines a linear ordering relationship among a set of keys
Unsorted Array Implementation

- **Insert**
  - increment size of array
  - put item at the end of the array
  - constant time: $O(1)$

- **Remove maximum**
  - find the max in the array by inspecting each element
  - exchange the maximum with the last item
  - decrement the size of the array
  - linear time: $O(N)$

*We can do better*

---

Heap

Imprecise Def'n: Storage of data in an array, such that each key is guaranteed to be larger than the key in two other specific positions

---

Heap-Ordered Trees, Heaps

Def'n: A tree is heap-ordered if the key in each node is $\geq$ the keys of all the node's children

Def'n: A heap is a set of nodes with keys arranged in a complete heap-ordered binary tree, represented as an array

Property: No node in a heap-ordered tree has a key larger than the key at the root
Pop Quiz (Analysis Question)

Given an array implementation of a heap, and the $i^{th}$ position in the array:
1) what is the location of $i$’s parent?
2) what is the location of $i$’s two children?

*assume that heap’s root is in position 1, not 0.*

- answer to 1) $\lfloor i/2 \rfloor$
- answer to 2) $2i$ and $2i + 1$

Breaking and Fixing a Heap

- What if priority of item on bottom of heap is increased?
  - need to *bottom-up heapify*
- What if priority of item on top of heap is decreased?
  - need to *top-down heapify*

Bottom Up Heapify

```cpp
void fixUp(Item heap[], int k)
{
    while (k > 1 && heap[k/2] < heap[k]){
        exch(heap[k], heap[k/2]);
        k = k/2;
    }
}
```

- Pass index (k) of array element w/ increased priority
- Exchange the key in the given node with the key of the parent until:
  - we reach the root, or
  - we reach a parent with a larger (or equal) key
- Note root is well-known (position 1)
Top Down Heapify

```c
void fixDown(Item heap[], int heapsize, int k)
    while (2*k <= heapsize)
        { int j = 2*k;
          if (j < heapsize && heap[j] < heap[j+1]) j++;
          if (heap[k] >= heap[j]) break;
          exch(heap[k], heap[j]); k = j;
        }
```

Pass index (k) of array element with decreased priority
- Exchange the key in the given node with the largest key among the node's children, moving down to that child, until:
  - we reach bottom of heap
  - there are no children with a larger key
- Unlike root, last node is not known in advance, must pass it (heapsize)

Heap Implementation

```c
void insert (Item item)
    { pq[++N] = item;
      fixUp(pq,N);
    }

Item getmax()
    { swap(pq[1], pq[N]);
      fixDown(pq, N-1, 1)
      return pq[N--];
    }
```

- Insert
  - put item at the end of the priority queue
  - use fixUp to find proper position
- Remove maximum
  - remove root
  - take item from end of array and place at root
  - use fixDown to find proper position

Properties of `insert` and `getmax`

Property: Insert requires no more than \( \lg N \) comparisons between heap elements
`insert`: \( O(\lg N) \)

Property: Find (and remove) max requires no more than 2 \( \lg N \) comparisons between heap elements
`getmax`: \( O(\lg N) \)
Intuition for Heapsort

- Repeatedly dequeue the highest priority element from a priority queue
- Advantages
  - easily implemented as an array
  - entire sort can be done in place

More Detail for Heapsort

- Phase 1
  - Transform unsorted array into heap
    • called ‘Heapifying’
- Phase 2
  - Remove the largest item from heap and add it to sorted sequence
  - Heapify
  - Repeat

Phase 1: Build Heap

```c
void buildHeap(Item heap[], int n)
{
    for (unsigned int i = n/2; i > 0; --i)
        fixDown(heap, n, i);
}
```

- Note order of node visitation in tree
- What would happen if algorithm started nearer to root?
Phase 2: Sort Heap

void sortHeap(Item heap[], int n)
{
    buildHeap(heap, n);
    for (unsigned int i = n; i >= 2; --i)
    {
        swap(heap[i], heap[1]);
        fixDown(heap, i-1, 1);
    }
}

- Remember first array index in heap is 1, not 0
- Make call to buildHeap
- Loop from last item in the heap
  - swap current root and current ‘last’ position
  - fix the heap

Heapsort

- Take N elements, insert into a heap
  - each takes at most O(log N) time, N of them
- Remove elements one at a time, filling original array from back to front.
  - each takes at most O(log N) time, N of them
- Total running time: O(N log N)
  - requires no additional space

Summary: Heaps

- Priority queue is an ADT
  - need insertion and removal
- Unsorted array
  - O(1) insertion of an item
  - O(N) removal of largest item
- Heap
  - efficient O(log N) insertion of an item
  - efficient O(log N) removal of largest item
- Must be able to maintain heap property
  - bottom-up heapify
  - top-down heapify
- Heapsort
  - O(N log N) sort that takes advantage of heap properties