COMP 2140 - Data Structures

Shahin Kamali

Topic 10 - Binary Trees via Arrays & Heaps
University of Manitoba

Partially based on notes by S. Durocher.
Overview

- Using an array instead of pointers to represent a binary tree
- Heaps via binary tree arrays
Arrays vs. Dynamic Allocation

- Typically, a tree is implemented using dynamic allocation of nodes. This is both:
  - more efficient with respect to memory usage, and
  - allows for faster operations.

- It is, however, also possible to implement a tree using an array. Practical only in restricted cases:
  - binary tree is complete
  - maximum size is known
  - only insert/delete rightmost leaf on bottom level
A complete binary tree is a binary tree in which every level is full, except possibly the bottom level. In the bottom level the nodes are in the leftmost positions.
Array Implementation of Binary Trees

- Simply write nodes level-by-level from left to right into an array.
Array Implementation of Binary Trees

When using an array to implement a complete tree:

- The root is stored at $A[0]$.
Array Implementation of Binary Trees

In general:

- The left child of the node at index i is stored at \( A[2i + 1] \).
- The right child of the node at index i is stored at \( A[2i + 2] \).
- The parent of the node at index i is stored at \( A[\lceil i/2 \rceil - 1] \).
Array Implementation of Binary Trees

- An array implementation can work for storing any tree.
Array Implementation of Binary Trees

- An array implementation can work for storing any tree
  - But if the tree is not complete or nearly complete, the memory will be wasted and the insert/delete take time as bad as $O(n)$.

```
  25
 /   \
22    17
/     / \
18 5  6 15
 /     /     \     \
8  22 6 15 9
```
Array Implementation of Binary Trees

- An array implementation can work for storing any tree
  - But if the tree is not complete or nearly complete, the memory will be wasted and the insert/delete take time as bad as $O(n)$.
- We often assume the array is complete and insert/delete take place at the last index $\rightarrow$ takes $O(1)$. 
A maximum binary heap is a complete binary tree such that the key of each node is less than or equal to the key of its parent, i.e., for any node $a$ in the tree, $\text{key}(a) \leq \text{key}(\text{parent}(a))$.

The heap property is recursive: each node’s left and right subtrees are also maximum binary heaps.
Binary Heaps

- A **maximum binary heap** is a complete binary tree such that the key of each node is less than or equal to the key of its parent, i.e., for any node \( a \) in the tree, \( \text{key}(a) \leq \text{key}(\text{parent}(a)) \).
  
  - The heap property is recursive: each node’s left and right subtrees are also maximum binary heaps.
  - If the opposite property holds then the data structure is a minimum binary heap.
Non-uniqueness of heaps

Consider the following two heaps:

- Both trees are complete binary trees.
- Both trees store the same keys.
- Both trees satisfy the maximum binary heap property.
- The positions of some keys differ in the two trees.
- This scenario is not possible for binary search trees.
Heap’s Application

- Heaps are used to implement priority queues.
- We want to maintain a collection of elements, each of which has a key corresponding to its priority:

  - `insert(key, data)`: insert a new item to the queue
  - `extractMax()`: remove and return the item with max key (priority)
  - `getMax()`: return the item with max key (priority)

```java
public interface PriorityQueueADT <E> {
    public void insert ( int key , E item );
    public E extractMax ();
    public E getMax ();
    public boolean isEmpty ();
    public int size ();
}
```
Heap’s Application

- Heaps are used to implement priority queues.
- We want to maintain a collection of elements, each of which has a key corresponding to its priority:
  - Insert(key, data): insert a new item to the queue
  - extractMax(): remove and return the item with max key (priority)
  - getMax(): return the item with max key (priority)

```java
public interface PriorityQueueADT<E> {
    public void insert(int key, E item);
    public E extractMax();
    public E getMax();
    public boolean isEmpty();
    public int size();
}
```
Heap’s Application

- Heaps are used to implement **priority queues**.
- We want to maintain a collection of elements, each of which has a key corresponding to its **priority**:
  - **Insert(key, data)**: insert a new item to the queue
  - **extractMax()**: remove and return the item with max key (priority)
  - **getMax()**: return the item with max key (priority)
  - other operations include isEmpty() and size();

```java
public interface PriorityQueueADT<E> {
    public void insert(int key, E item);
    public E extractMax();
    public E getMax();
    public boolean isEmpty();
    public int size();
}
```
Heaps are used to implement priority queues.

We want to maintain a collection of elements, each of which has a key corresponding to its priority:

- **Insert(key, data)**: insert a new item to the queue
- **extractMax()**: remove and return the item with max key (priority)
- **getMax()**: return the item with max key (priority)
- other operations include isEmpty() and size();

```java
public interface PriorityQueueADT<E> {
    public void insert(int key, E item);
    public E extractMax();
    public E getMax();
    public boolean isEmpty();
    public int size();
}
```

We will show how to use a heap to implement these operations.
An instance of HeapElem is a pair (data, key), like a node in the tree.

A heap is basically an array of HeapElem objects.

```java
public class HeapElem<E> {
    public int key;
    public E data;
    public HeapElem(int newKey, E newData) {
        key = newKey;
        data = newData;
    }
}
```
Heap Implementation

- An instance of HeapElem is a pair (data, key), like a node in the tree.
- A heap is basically an array of HeapElem objects.

```java
public class HeapElem<E> {
    public int key;
    public E data;
    public HeapElem(int newKey, E newData) {
        key = newKey;
        data = newData;
    }
}

public class Heap<E> implements PriorityQueueADT<E> {
    private HeapElem<E>[] heap;
    private int size, maxSize;
    public Heap(int newSize) {
        maxSize = newSize;
        heap = new HeapElem<E>[maxSize];
        size = 0;
    }
```
Insertion in Heaps

- A new element is inserted as the next empty leaf in the complete binary tree.

- A `reheapUp` operation is performed on the new node:
  - compare the new item with its parent
  - if its key is larger than its parent’s key, swap the two nodes
  - continue recursively upwards

- E.g.: `insert(E)`
Insertion in Heaps

- A new element is inserted as the next empty leaf in the complete binary tree.
- A `reheapUp` operation is performed on the new node:
  - compare the new item with its parent
  - if its key is larger than its parent’s key, swap the two nodes
  - continue recursively upwards
- E.g.: `insert(E)`
Insertion in Heaps

- A new element is inserted as the next empty leaf in the complete binary tree.

- A \textbf{reheapUp} operation is performed on the new node:
  - compare the new item with its parent
  - if its key is larger than its parent’s key, swap the two nodes
  - continue recursively upwards

- E.g.: insert(E)
Insertion in Heaps

- A new element is inserted as the next empty leaf in the complete binary tree.
- A *reheapUp* operation is performed on the new node:
  - compare the new item with its parent
  - if its key is larger than its parent’s key, swap the two nodes
  - continue recursively upwards
- E.g.: insert(E)
Insertion into a Heap

• insert is a public method that calls the private reheapUp

```java
public void insert(int newKey, E item) {
    if (size < maxSize) {
        heap[size] = new HeapElem<E>(newKey, item);
        reheapUp(size);
        size++;
    }
}

private void reheapUp(int index) {
    int parent = (index + 1) / 2 - 1;
    if (index > 0 && heap[parent].key < heap[index].key) {
        swap(parent, index);
        reheapUp(parent);
    }
}
```
ExtractMax

- We want to Remove the item of highest priority and return it.
- An item of highest priority is always located at the root of the tree.
  - Copy the item at the root to return it later.
  - Take the rightmost element on the bottom level of the tree (the last item currently in the array) and move it to the root. This preserves the structure of the complete binary tree but the heap ordering property is lost.
  - Perform a reheapDown operation on the root.
  - Return the maximum item that was stored.
We want to Remove the item of highest priority and return it.

An item of highest priority is always located at the root of the tree.

- Copy the item at the root to return it later.
- Take the rightmost element on the bottom level of the tree (the last item currently in the array) and move it to the root. This preserves the structure of the complete binary tree but the heap ordering property is lost.
- Perform a reheapDown operation on the root.
- Return the maximum item that was stored.
ExtractMax

- We want to remove the item of highest priority and return it.
- An item of highest priority is always located at the root of the tree.
  - Copy the item at the root to return it later.
  - Take the rightmost element on the bottom level of the tree (the last item currently in the array) and move it to the root. This preserves the structure of the complete binary tree but the heap ordering property is lost.
  - Perform a reheapDown operation on the root.
  - Return the maximum item that was stored.
ExtractMax

- We want to remove the item of highest priority and return it.
- An item of highest priority is always located at the root of the tree.
  - Copy the item at the root to return it later.
  - Take the rightmost element on the bottom level of the tree (the last item currently in the array) and move it to the root. This preserves the structure of the complete binary tree but the heap ordering property is lost.
  - Perform a reheapDown operation on the root.
- Return the maximum item that was stored.
ExtractMax

- We want to Remove the item of highest priority and return it.
- An item of highest priority is always located at the root of the tree.
  - Copy the item at the root to return it later.
  - Take the rightmost element on the bottom level of the tree (the last item currently in the array) and move it to the root. This preserves the structure of the complete binary tree but the heap ordering property is lost.
  - Perform a reheapDown operation on the root.
  - Return the maximum item that was stored.
public E extractMax() {
    E result = null;
    if (size > 0) {
        result = heap[0].data;
        heap[0] = heap[--size];
        reheapDown(0);
    }
    return result;
}
Extracting the max item from a Heap

```java
public E extractMax() {
    E result = null;
    if (size > 0) {
        result = heap[0].data;
        heap[0] = heap[--size];
        reheapDown(0);
    }
    return result;
}

private void reheapDown (int top) {
    int maxChild;
    int left = 2 * top + 1;
    int right = 2 * top + 2;
    if (left < size) {
        if (right >= size || heap[left].key > heap[right].key)
            maxChild = left;
        else maxChild = right;
        if (heap[top].key < heap[maxChild].key) {
            swap(top, maxChild);
            reheapDown(maxChild);
        }
    }
}
```
What is the time Complexity of ReheapUp and ReheapDown?

When performing either a reheapUp or reheapDown operation, the number of steps depends on the **height** of the tree.

- In the worst case a single path from root to leaf is traversed.

A complete binary tree of height $h$ has between $2^h$ and $2h + 11$ nodes

- Therefore, a complete binary tree with $n$ nodes has height between $\log(n + 1) - 1$ and $\log n$
- Hence, the **time complexity of reheapUp and reheapDown** is $O(\log n)$. 

---

**Time Complexity**

- What is the time Complexity of ReheapUp and ReheapDown?
- When performing either a reheapUp or reheapDown operation, the number of steps depends on the **height** of the tree.
  - In the worst case a single path from root to leaf is traversed.
- A complete binary tree of height $h$ has between $2^h$ and $2h + 11$ nodes
  - Therefore, a complete binary tree with $n$ nodes has height between $\log(n + 1) - 1$ and $\log n$
  - Hence, the **time complexity of reheapUp and reheapDown** is $O(\log n)$. 

---

**COMP 2140 - Data Structures**
what is time Complexity of Insert and ExtractMax?

- Inserting an item into the priority queue requires one call to reheapUp which takes \( O(\log n) \) time.
- Removing the maximum item from the priority queue requires one call to reheapDown which takes \( O(\log n) \) time.
- Insert and Extract-Max operations in a heap take \( O(\log n) \) time.
what is time Complexity of Insert and ExtractMax?

Inserting an item into the priority queue requires one call to reheapUp which takes $O(\log n)$ time.

Removing the maximum item from the priority queue requires one call to reheapDown which takes $O(\log n)$ time.
what is time Complexity of Insert and ExtractMax?

Inserting an item into the priority queue requires one call to reheapUp which takes $O(\log n)$ time.

Removing the maximum item from the priority queue requires one call to reheapDown which takes $O(\log n)$ time.

Insert and Extract-Max operations in a heap take $O(\log n)$ time.
Given an arbitrary array, how can we make it into a heap? Similarly, given an arbitrary complete binary tree, how can we make it into a heap?

Solution:
Start at the bottom of the tree to restore the heap property within each subtree and work upwards towards the root.

```java
public void buildHeap () {
    for (int i = size - 1; i >= 0; i--)
        reheapDown(i);
}
```
Building a Heap

Given an arbitrary array, how can we make it into a heap? Similarly, given an arbitrary complete binary tree, how can we make it into a heap?

**Solution:** Start at the bottom of the tree to restore the heap property within each subtree and work upwards towards the root.

```java
public void buildHeap() {
    for (int i = size - 1; i >= 0; i--)
        reheapDown(i);
}
```
BuildHeap (Heapify) Example

Convert the following array (complete binary tree) into a heap:

\[ \text{reheapDown}(7) \]

![Binary tree diagram]
BuildHeap (Heapify) Example

Convert the following array (complete binary tree) into a heap:

reheapDown(6)
BuildHeap (Heapify) Example

Convert the following array (complete binary tree) into a heap:

reheapDown(5)
Convert the following array (complete binary tree) into a heap:

reheapDown(4)
BuildHeap (Heapify) Example

Convert the following array (complete binary tree) into a heap:

reheapDown(3)

[Diagram of a complete binary tree with nodes labeled 4, 6, 1, 3, 2, 5, 0, 9]

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>0</td>
<td>9</td>
</tr>
</tbody>
</table>
BuildHeap (Heapify) Example

Convert the following array (complete binary tree) into a heap:

reheapDown(3)
Convert the following array (complete binary tree) into a heap:

reheapDown(2)
Convert the following array (complete binary tree) into a heap:

```
reheapDown(2)
```

![Binary tree diagram and array representation]
BuildHeap (Heapify) Example

Convert the following array (complete binary tree) into a heap:

reheapDown(1)
Convert the following array (complete binary tree) into a heap:

reheapDown(1)
BuildHeap (Heapify) Example

Convert the following array (complete binary tree) into a heap:

reheapDown(0)
Convert the following array (complete binary tree) into a heap:

reheapDown(0)
BuildHeap (Heapify) Example

Convert the following array (complete binary tree) into a heap:

reheapDown(0)
Reheap Time Complexity

- A call to `reheapDown` takes time $O(\log n)$ in the worst case.
- `buildHeap` makes $n$ calls to `reheapDown`.
- Therefore, the runtime of `buildHeap` is $O(n \log n)$.
Reheap Time Complexity

- A call to `reheapDown` takes time $O(\log n)$ in the worst case.
- `buildHeap` makes $n$ calls to `reheapDown`.
- Therefore, the runtime of `buildHeap` is $O(n \log n)$.
- This is true, but we can give a better bound. In fact, the runtime of `buildHeap` is $O(n)$. 
Reheap Time Complexity

- A call to `reheapDown` takes time $O(\log n)$ in the worst case.
- `buildHeap` makes $n$ calls to `reheapDown`.
- Therefore, the runtime of `buildHeap` is $O(n \log n)$.
- This is true, but we can give a better bound. In fact, the runtime of `buildHeap` is $O(n)$.
- **Given an array of numbers, we can form a heap from them in time $O(n)$.*
Heap Summary

- Heap is a **simple** data structure that implements priority queues.
- `extract-max` can be implemented using `reheapDown` in $O(\log n)$.
- `insert` can be implemented using `reheapUp` in $O(\log n)$.
- Given an array of keys, it is possible to build a heap from them (heapify) in $O(n)$. 