

# 5.6

## Heaps

2018/10/22 © Ren-Song Tsay, NTHU, Taiwan 44

---

---

---

---

---

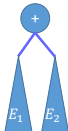

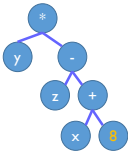
---

---

---

### Expression Tree

- Given a regular expression, put **operands** at **leaf** nodes and **operators** at **nonterminal** nodes

Inorder	$E1 + E2$	$a + b$	$y * (z - (x + 8))$	Infix notation
Preorder	$+ E1 E2$	$+ a b$	$* y - z + x 8$	Prefix notation
Postorder	$E1 E2 +$	$a b +$	$y z x 8 + - *$	Postfix notation

---

---

---

---

---

---

---

---

5.6.1

### Priority Queue

- In a **priority queue**, the element to be processed/deleted is the one with the **highest** (or **lowest**) priority
- Operations
  - Get the max/min element
  - Insert an element to the priority queue
  - Delete element with max/min priority

46

---

---

---

---

---

---

---

---

## ADT: Priority Queue

```

template < class T >
class MaxPQ
{
public:
    MaxPQ();
    ~MaxPQ();

    // Check if PQ is empty
    bool IsEmpty() const;
    // Return reference to the max element
    T& Top() const;
    // Add an element to the PQ
    void Push(const T&);
    // Delete element with max priority
    void Pop();
private:
    // Data representation here
    // ...
};
    
```

---

---

---

---

---

---

---

---

---

---

## PQ Representations

- Unsorted linear list
  - Array, chain, ..., etc.
- Sorted linear list
  - Sorted array, sorted chain, ..., etc.
- Heap

	Top()	Push()	Pop()
Unsorted linear list	$O(n)$	$O(1)$	$O(n)$
Sorted linear list	$O(1)$	$O(n)$	$O(1)$
Heap	$O(1)$	$O(\log n)$	$O(\log n)$

---

---

---

---

---

---

---

---

---

---

5.6.2

## Max Heap Definition

- A **max (min) tree** is a tree in which the key value in each node is **no smaller (larger)** than the key values in its children (if any).
  - A **max(min) heap** is a **complete binary tree** that is also a **max(min) tree**.




---

---

---

---

---

---

---

---

---

---

5.6.2 **Non-heap**

Not a heap  
(12 > 10)

Not a heap  
(Not a complete binary tree)

50

---

---

---

---

---

---

---

---

---

---

**Max Heap: Representation**

- Since the heap is a complete binary tree, we may adopt “**Array Representation**” as mentioned before!
- Let node  $i$  be in position  $i$  (array[0] is empty)
  - $Parent(i) = \lfloor i/2 \rfloor$ , if  $i \neq 1$ . If  $i = 1$ ,  $i$  is the root and has no parent.
  - $leftChild(i) = 2i$ , if  $2i \leq n$ . If  $2i > n$ , the  $i$  has no left child.
  - $rightChild(i) = 2i + 1$ , if  $2i + 1 \leq n$ , if  $2i + 1 > n$ , the  $i$  has no right child.

---

---

---

---

---

---

---

---

---

---

**ADT: Priority Queue**

```

template < class T >
class MaxPQ
{
public:
    MaxPQ();
    ~MaxPQ();

    // Check if PQ is empty
    bool IsEmpty() const;
    // Return reference to the max element
    T& Top() const;
    // Add an element to the PQ
    void Push(const T&);
    // Delete element with max priority
    void Pop();
private:
    T* heap; // Element array
    int heapSize; // # of elements
    int capacity; // size of the array "heap"
};
    
```

---

---

---

---

---

---

---

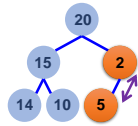
---

---

---

### Max Heap: Insert

- Insert a node with key value = 5
- Make sure it is a complete binary tree
- Check if the new node is greater than its parent
- If so, swap two nodes



53

---

---

---

---

---

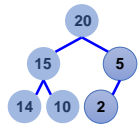
---

---

---

### Max Heap: After Insertion

- Insert a node with key value = 5
- Make sure it is a complete binary tree
- Check if the new node is greater than its parent
- If so, swap two nodes



54

---

---

---

---

---

---

---

---

### Max Heap: Insert Code

```

template < class T >
void MaxPQ<T>::Push(const T& e)
{ // Insert e into max heap
  // Make sure the array has enough space here...
  // ...
  int currentNode = ++heapSize;
  while(currentNode != 1 && heap[currentNode/2] < e)
  { // Swap with parent node
    heap[currentNode]=heap[currentNode/2];
    currentNode /= 2; // currentNode now points to parent
  }
  heap[currentNode]=e;
}
    
```

**Time Complexity**  
 Travel at most the height of a tree, therefore is  $O(\log n)$

---

---

---

---

---

---

---

---

### Max Heap: Delete

1. Always delete the root
2. Move the last element to the root (maintain a complete binary tree)

56

---

---

---

---

---

---

---

---

### Max Heap: Delete

1. Always delete the root
2. Move the last element to the root ( maintain a complete binary tree )
3. Swap with the largest child (if any)

57

---

---

---

---

---

---

---

---

### Max Heap: Delete

1. Always delete the root
2. Move the last element to the root ( maintain a complete binary tree )
3. Swap with the largest child (if any)
4. Continue step 3 until the max heap is maintained (trickle down)

58

---

---

---

---

---

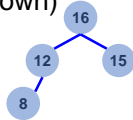
---

---

---

## Max Heap: Delete

1. Always delete the root
2. Move the last element to the root ( maintain a complete binary tree )
3. Swap with the largest child (if any)
4. Continue step 3 until the max heap is maintained (trickle down)



59

---

---

---

---

---

---

---

---

## Max Heap: Delete Codes

```

template < class T >
void MaxPQ<T>::Pop()
{ //Delete max element
  if(IsEmpty()) throw "Heap is empty";
  heap[1].~T(); // delete max element (always the root!)
  // Remove the last element from heap
  T lastE = heap[heapSize--];

  // trickle down
  int currentNode = 1; // root
  int child = 2; // A child of currentNode
  while(child <= heapSize) {
    // Set child to larger child of currentNode
    if (child < heapSize && heap[child] < heap[child + 1]) child++;

    // Can we put lastE in currentNode?
    if (lastE >= heap[child]) break; // Yes!

    // No!
    heap[currentNode] = heap[child]; // Move child up
    currentNode = child; child *=2; // Move down a level
  }
  heap[currentNode] = lastE;
}
    
```

**Time Complexity = Height of tree =  $O(\log n)$**

---

---

---

---

---

---

---

---