Algorithms – I (CS29003/203)

Autumn 2022, IIT Kharagpur

Priority Queues and Heaps

Breakthrough during the Vacation

Demis Hassabis
@demishassabis

Since 1969 Strassen's algorithm has famously stood as the fastest way to multiply 2 matrices - but with #AlphaTensor we've found a new algorithm that's faster, with potential to improve efficiency by 10-20% across trillions of calculations per day!

deepmind.com
Discovering novel algorithms with AlphaTensor
In our paper, published today in Nature, we introduce AlphaTensor, the first artificial intelligence (AI) system for discovering novel, efficient, and provably ...

10:30 PM · Oct 5, 2022 · Twitter Web App

Resources

- Apart from the book
- UC Davis ECS 36C Course by Prof. Joël Porquet-Lupine

Introduction

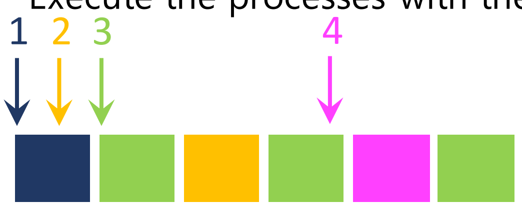
- Let us think about a scenario where you are performing the following tasks simultaneously
  - Start a long code compilation
  - Refresh moodle page
  - Send Gmail chat message
  - Receive email notification
- Objective: Execute multiple processes until completion, but keep the system responsive

- Are processes executed until completion?
- Will the system feel responsive?

Process Scheduling

- Round-robin scheduling
  - Add processes to a (typically FIFO) queue
  - Execute them for equal chunks of time

  - Will the system feel more responsive? Can we do better?
- Priority scheduling
  - Associate priority with each process, e.g., short processes get higher priority
  - Execute the processes with the highest priority first

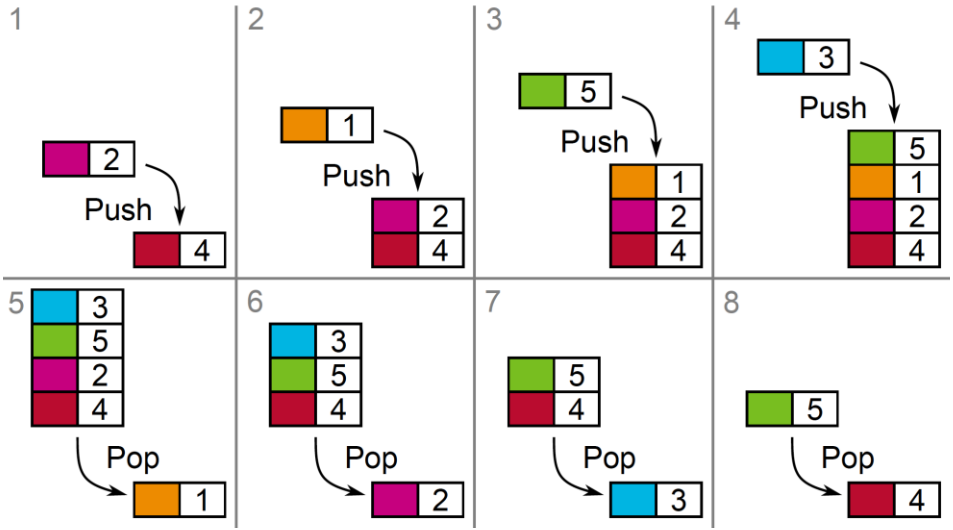


A very good resource on process scheduling:
[Prof. Krzyzanowski's lecture notes]

Source: UC Davis, ECS 36C course, Spring 2020

Priority Queue

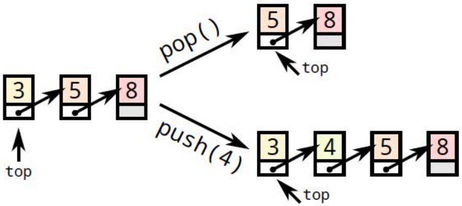
- Definition: **Queue** where each item is associated to a priority (i.e., a comparable key)
  - **Push/Insert**: add item and associated priority
  - **Pop/Pull**: remove item with highest priority



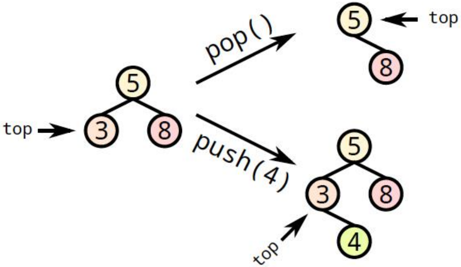
Source: UC Davis, ECS 36C course, Spring 2020

Naïve Implementations

- Sorted list
  - Keep sorted by priority when pushing
  - Pop highest priority item from front



- Binary Search Tree (self-balanced)
  - Restructure tree by priority when inserting
  - Return highest priority item from minimum of tree



Source: UC Davis, ECS 36C course, Spring 2020

List Implementations

- Struct node

```
// Node
struct node {
  // Lower values indicate higher priority
  int priority;
  struct node* next;
};
```

- Function to create a new node

```
// Function to Create A New Node
struct node* newNode(int p)
{
  struct node* temp = (struct node*)malloc(sizeof(struct node));
  temp->priority = p;
  temp->next = NULL;
  return temp;
}
```

- Function to check if queue is empty

```
bool isEmpty(struct node* head)
{
  return head == NULL;
}
```

List Implementations

- Get value of the maximum priority (minimum element)

```
int getMin(struct node* head)
{
  if(isEmpty(head))
  {
    printf("Empty priority queue!\n");
    exit(0);
  }
  return head->priority;
}
```

- Extract the element with maximum priority (minimum element)

```
struct node* pop(struct node* head){
  if(isEmpty(head))
  {
    printf("Empty priority queue!\n");
    exit(0);
  }
  struct node* temp = head;
  head = head->next;
  free(temp);
  // Return new head
  return head;
}
```

List Implementations

- Push new element at the right place

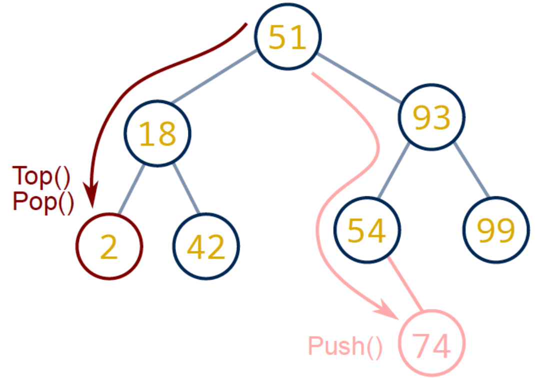
```
struct node* push(struct node* head, int newNum){
  // Create new Node
  struct node* nNode = newNode(newNum);
  // The head of list has lesser priority than new node
  if(head->priority>newNum)
  {
    nNode->next = head;
    // Return new head
    return nNode;
  }
  else
  {
    // Traverse the list and find the position
    // to insert the new node
    struct node* start = head;
    struct node* startPrev = head;
    while(newNum>start->priority)
    {
      if(start->next!=NULL)
      {
        startPrev = start;
        start = start->next;
      }
      else
      {
        // It has reached the end of list. Insert the new node
        start->next = nNode;
        // Return head
        return head;
      }
    }
    // Store what is now pointed by 'next' field of the node whose
    // priority is just above it (previous node of current start)
    nNode->next = startPrev->next;
    // Connect 'next' field of previous node to newNode's next
    startPrev->next = nNode;

    // Return head
    return head;
  }
}
```

List Implementations

- Complexities?
  - getMin() - $O(1)$
  - pop() - $O(1)$
  - push() - $O(n)$

Binary Search Tree Implementations

- A priority queue can also be entirely mapped onto a BST implementation
- getMin()
  - Use getMin() - $O(\log n)$
- pop()
  - Use delete(getMin()) - $O(\log n)$
- push()
  - Use insert() method of BST - $O(\log n)$

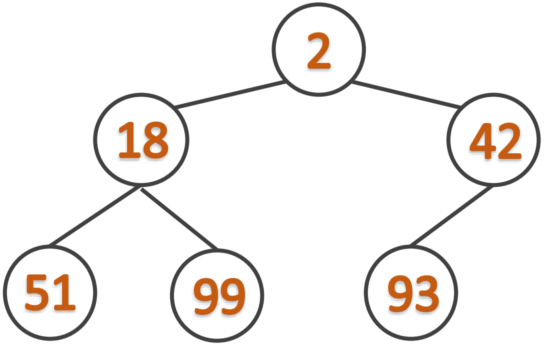


- Conclusion on naïve implementations
  - A list implementation gives getMin()/pop() in $O(1)$
    - But push() in $O(n)$
  - A BST implementation gives push() in $O(\log n)$
    - But getMin()/pop() in $O(\log n)$ as well
  - How to get the best of both worlds?

Source: UC Davis, ECS 36C course, Spring 2020

Binary Heap

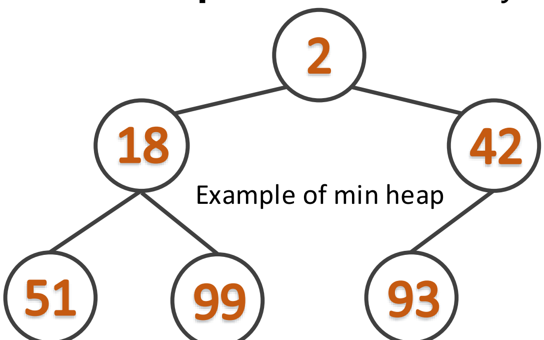
- Most common data structure for implementing a priority queue
- So ubiquitous in implementing priority queues that the word '*heap*' is used without any qualifier in this context
- A heap is a binary tree with two properties
  - Heap-order property
  - Structure property
- Operations on heaps can destroy one or more of these properties
- So a heap operation must not terminate until all heap properties are in order



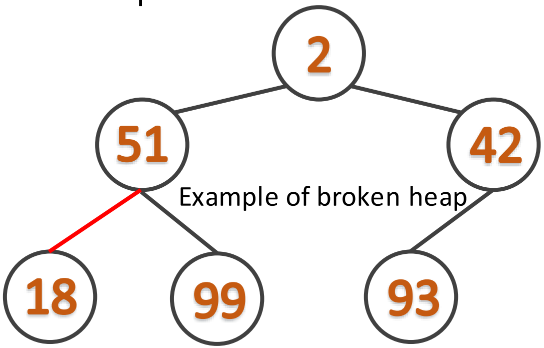
Source: UC Davis, ECS 36C course, Spring 2020

Heap-Order Property

- Since we want to find the minimum quickly, it makes sense that the smallest element should be at the root.
- Continuing, any node should be smaller than all of its descendants
- Giving two types of Heaps
  - **min-heaps**: each node's key is less than or equal to each of its children
  - **max-heaps**: each node's key is greater than or equal to each of its children



Example of min heap

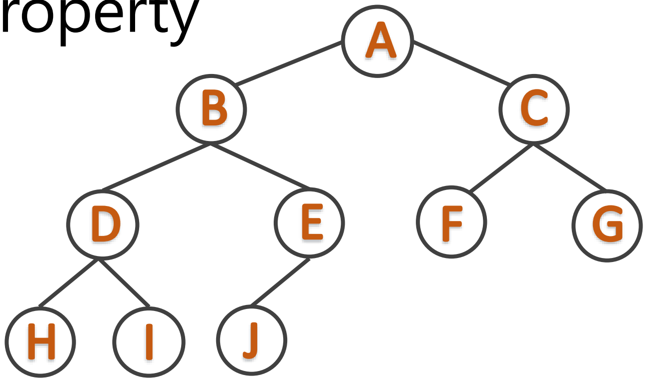


Example of broken heap

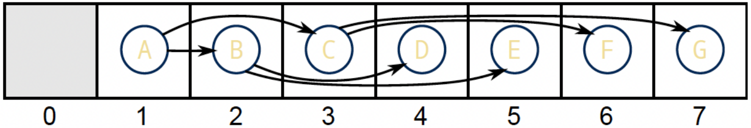
- By design, item with highest priority is always the root
  - getMin() is $O(1)$

Source: UC Davis, ECS 36C course, Spring 2020

Structure Property



- A heap is a complete binary tree
  - All levels are completely filled, apart from possibly the last
  - The last level is packed to the left
- Guarantee of height in $O(\log n)$
- Easy representation
  - Complete binary tree is so regular, it can be represented in an array
  - No need for complicated link management, and fast traversal



- Root is always at index 1

For node at index i*:
- Left child at index $2i$
- Right child at index $2i+1$
- Parent (if not root) at index $\lfloor \frac{i}{2} \rfloor$

*Heap could start at array index 0, but node indexing would become a little more complex

Source: UC Davis, ECS 36C course, Spring 2020

Implementation

- Min-heap version
- Simple array (capacity incremented by 1 for the unused first element)

```
#define CAPACITY 10

// Global variable to track the size of Heap
int size = 0;
```

```
// Remember we are starting at index 1
int H[CAPACITY+1];
```

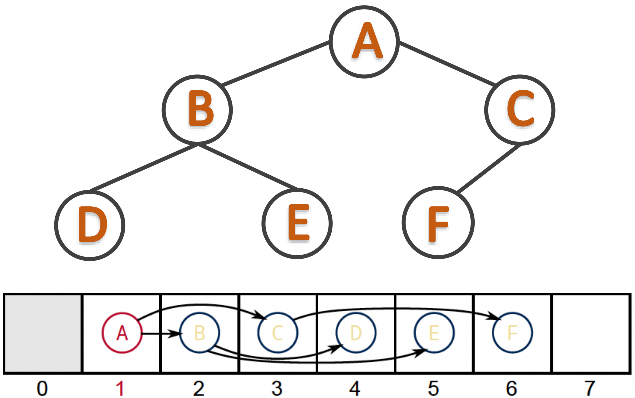
```
// Helper methods for indices
int Root(){
return 1;
}
int Parent(int n){
return n/2;
}
int LeftChild(int n){
return 2*n;
}
int RightChild(int n){
return 2*n + 1;
}
```

```
// Helper methods for node testing
bool HasParent(int n){
  return n!=Root();
}
bool IsNode(int n){
  return n<=size;
}
```

Source: UC Davis, ECS 36C course, Spring 2020

Implementation

- getMin() – Returns the node with highest priority (minimum value in min-heap)
- It is as simple as returning the root only



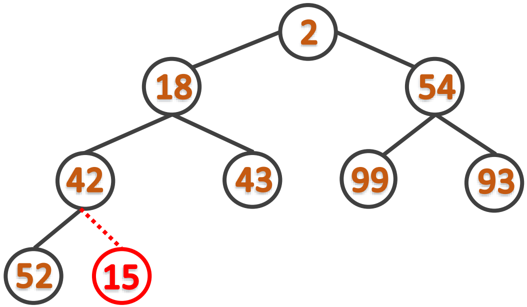

```c
// Function to get the node with highest
// priority minimum element (root)
int getMin(int H[])
{
    if(size == 0)
    {
      printf("Empty priority queue!\n");
      exit(0);
    }
    return H[Root()];
}
```

- Next, we will insert or push() an element into the heap
- What is the most natural position for the new element? (Don't worry about the heap order property, just keep the structure property intact)

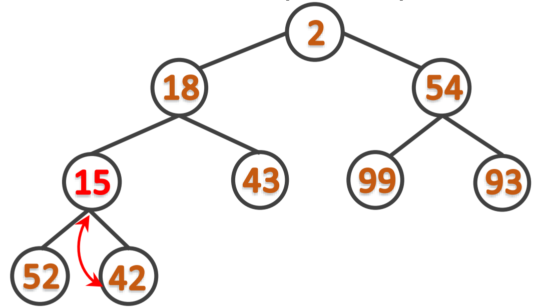
Source: UC Davis, ECS 36C course, Spring 2020

Implementation

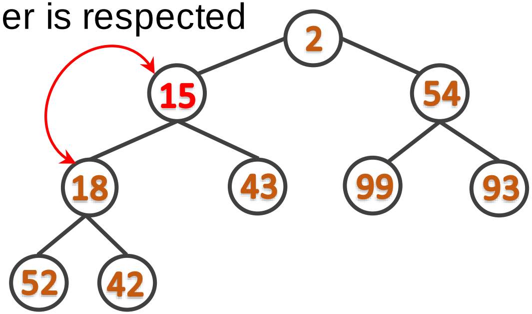
- push() – Insert only at locations that keeps the tree complete



- If heap-order not broken, stop!
- Otherwise, swap with parent



- Continue going up tree until heap-order is respected



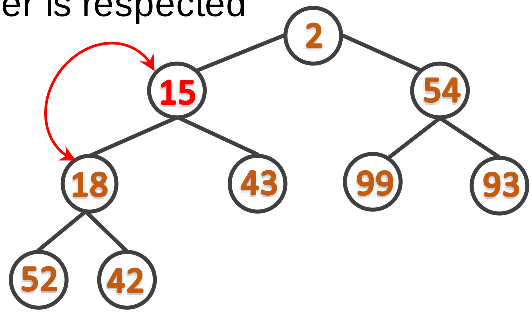
- Strategy known as shift up
- Also called bubble up, heapify-up

```
// Function to shift up the node in order
// to maintain the heap property
void shiftUp(int H[], int n){
  while (HasParent(n) && (H[Parent(n)] > H[n]))
  {
    swap(&H[Parent(n)], &H[n]);
    // Inside the while loop change
    // n to go to its parent
    n = Parent(n);
  }
}
```

Source: UC Davis, ECS 36C course, Spring 2020

Implementation

- Continue going up tree until heap-order is respected



- Strategy known as shift up
- Also called bubble up, heapify-up

```
// Function to shift up the node in order
// to maintain the heap property
void shiftUp(int H[], int n){
  while (HasParent(n) && (H[Parent(n)] > H[n]))
  {
    swap(&H[Parent(n)], &H[n]);
    // Inside the while loop change
    // n to go to its parent
    n = Parent(n);
  }
}
```

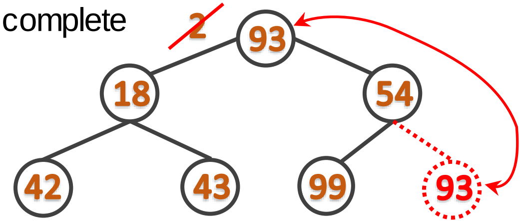
- Insertion of new item

```
void push(int H[], int newNum){
  // Check if heap is full
  if(size == CAPACITY)
  {
    printf("Priority queue full!\n");
    exit(0);
  }
  // Insert at the end
  H[size+1] = newNum;
  size++;
  // Shift up
  shiftUp(H, size);
}
```

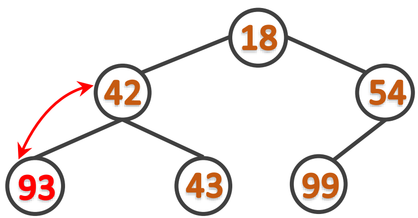
Source: UC Davis, ECS 36C course, Spring 2020

Implementation

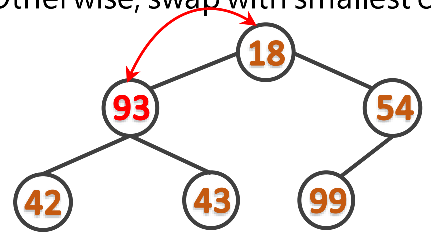
- pop() – Removing min item leaves hole at root. Move last item to root to keep the tree complete



- Continue going down tree until heap-order is respected again



- If heap-order not broken, stop!
- Otherwise, swap with smallest child



- Strategy known as shift down
- Also called bubble down, heapify-down

Source: UC Davis, ECS 36C course, Spring 2020

Implementation

- Strategy known as shift down
- Also called bubble down, heapify-down

```
// Function to shift down the node in order
// to maintain the heap property
void shiftDown(int H[], int n){
  // While node has at least one child
  // (if one, necessarily on the left)
  while(IsNode(LeftChild(n)))
  {
    // Consider left child by default
    int child = LeftChild(n);
    // If right child exists and smaller than
    // left child, then consider right child
    if(IsNode(RightChild(n)) && (H[RightChild(n)] < H[LeftChild(n)]))
      child = RightChild(n);
    // Exchange smallest child with node to
    // restore heap-order if necessary
    if(H[n]>H[child])
      swap(&H[n], &H[child]);
    else
      break;

    // Inside the while loop change
    // n to go one level down
    n = child;
  }
}
```

```
// Function to extract the element with
// maximum priority (in min-heap, this is
// node with the min value)
void Pop(int H[]){
  if(size == 0)
  {
    printf("Empty priority queue!\n");
    exit(0);
  }
  // Move last item back to root and reduce
  // heap's size
  H[Root()] = H[size];
  size--;
  shiftDown(H, Root());
}
```

Conclusion

- Running time complexities

Push	Pop	Top
$O(\log n)$	$O(\log n)$	$O(1)$

- Other operations:
  - In-place item modification, e.g., when heap/priority queue is used for process scheduling, change priority of a process
  - DecreaseItem(Item, DeltaPriority)
  - IncreaseItem(Item, DeltaPriority)
- Heap variants:
  - 2-3 heap, Binomial heaps, d-ary heaps, Fibonacci heaps, Leftist heaps, Skew heaps etc.

Type	Push	Pop	Top
Sorted list	$O(n)$	$O(1)$	$O(1)$
BST	$O(\log n)$	$O(\log n)$	$O(\log n)$
Binary heap	$O(\log n)$	$O(\log n)$	$O(1)$

Source: UC Davis, ECS 36C course, Spring 2020

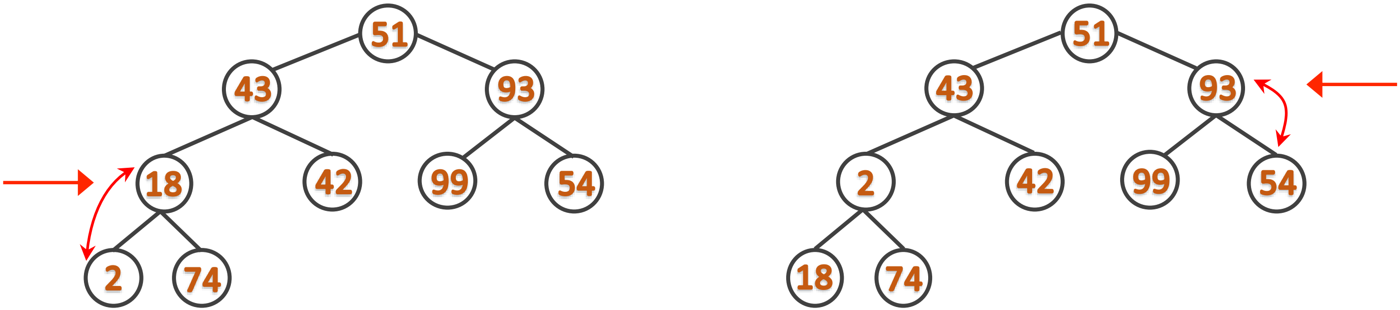
Heapify/BuildHeap

- Sometimes, build an entire heap directly out of an initial collection of items
- One use-case is Heapsort – an efficient way to sort an array
- Naïve approach (also known as Williams' method)
  - For all N items, push() N times
- Each push() operation takes $O(1)$ average time and $O(\log N)$ worst-case time
- For N items, this algorithm will run in $O(N)$ average time and $O(\mathrm{N}\log N)$ worst-case time
- Is there a better way?

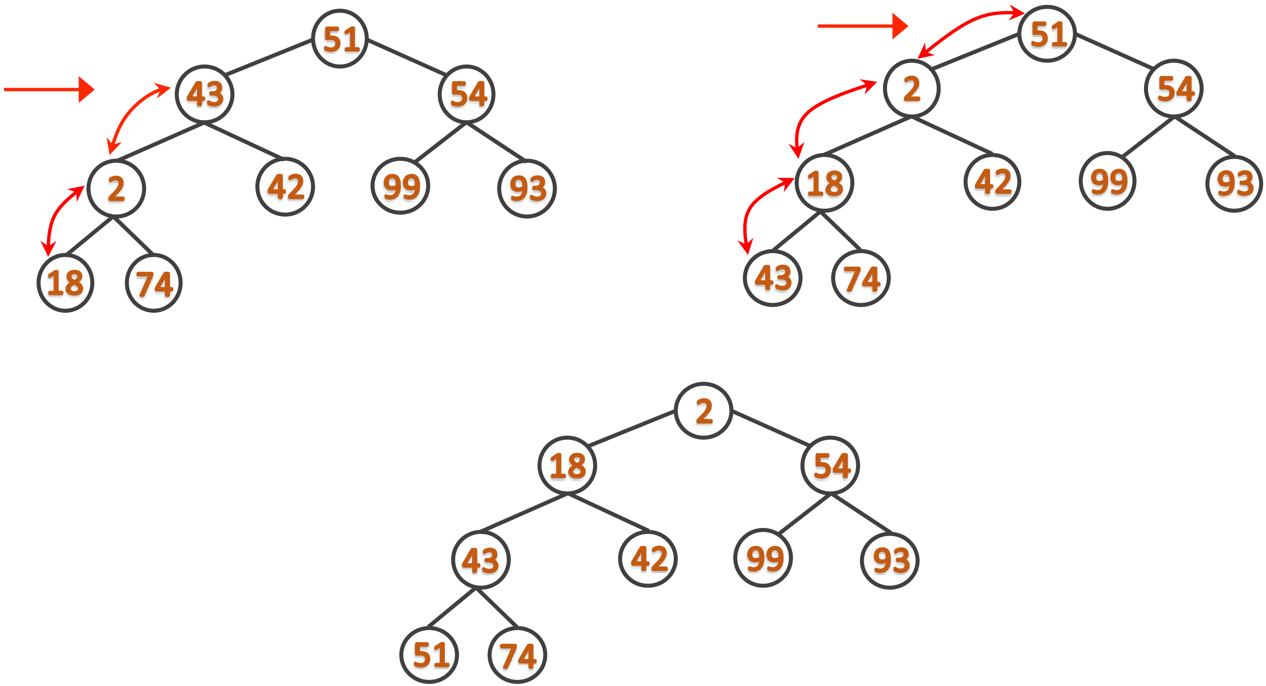
Heapify/BuildHeap

- Linear approach (Floyd's method)
  - Put all the elements on binary tree, only respecting structure property
  - Shift down all the elements starting from the first node who has at least one child, and up to the root

```
// Array
int arr[] = {51, 43, 93, 18, 42, 99, 54, 2, 74};
```



Heapify/BuildHeap



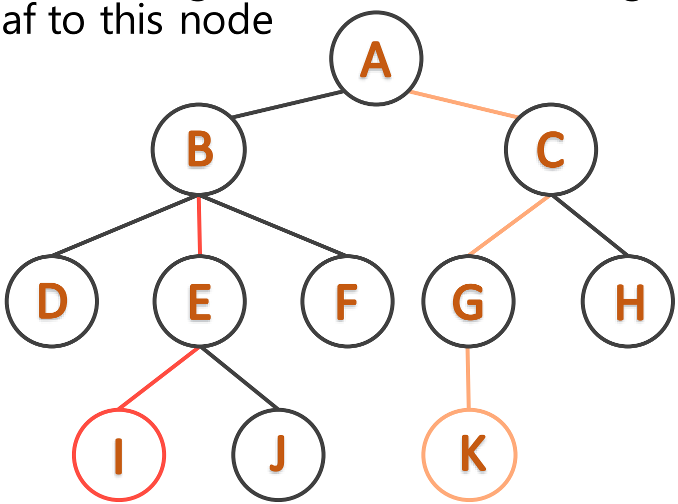
Heapify/BuildHeap Implementations

```
// Main code
int main()
{
  // Array
  int arr[] = {51, 43, 93, 18, 42, 99, 54, 2, 74};
  // Size of array
  int n = sizeof(arr)/sizeof(arr[0]);
  // Print the array
  printf("Array: ");
  for (int i = 0; i < n; i++)
    printf("%d ", arr[i]);
  printf("\n");

  // Remember we are starting at index 1
  int H[CAPACITY+1];
  // Building heap
  buildheap(arr, H, n);
  // Print after heapifying
  printf("Min-Heap: ");
  printHeap(H, size);

  return 0;
}
```

```
void buildheap(int arr[], int H[], int n){
  // Only enforce structure property
  for(int i = 0; i < n; i++)
    H[i+1] = arr[i];
  // Set the global variable size's value
  size = n;

  // Now enforce heap-property
  for (int i = size/2; i >= 1; i--)
    shiftDown(H, i);
}
```

Heapify/BuildHeap Complexity Analysis

- The running time of heapify operation is $\Theta(n)$ where n is the number of elements in the heap
- RECAP - <u>Height</u>: The height of a node is the length of the longest path from a leaf to this node



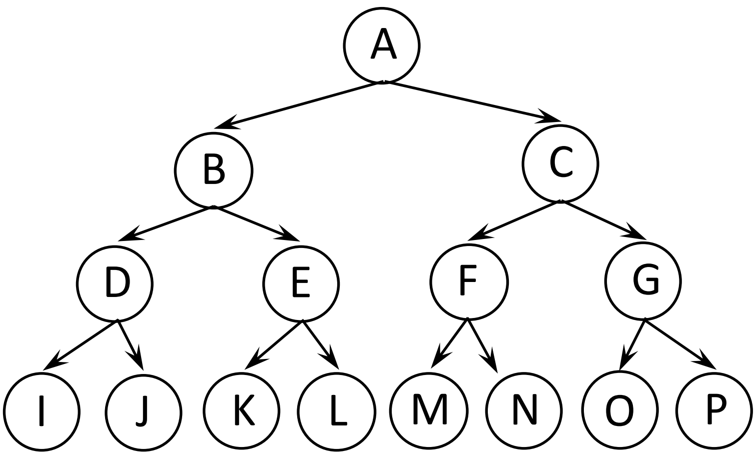
- All leaves are at height 0
- The height of a tree is equal to the height of the root from the deepest leaf (which is always equal to the depth of the tree)

Heapify/BuildHeap Complexity Analysis

- The running time of heapify operation is $\Theta(n)$ where n is the number of elements in the heap
- Some intuitions first [Great resource on this topic - https://stackoverflow.com/a/18742428]
- The basic idea is after creating a complete binary tree - move an offending node until it satisfies the heap property
  - shiftUp - swaps a node that is less than its parent (thereby moving it up) until it is no smaller than the node above it
  - shiftDown - swaps a node that is more than its smallest child (thereby moving it down) until it is no larger than both nodes below it
- The number of operations required for shiftUp and shiftDown is proportional to the distance the node may have to move
- For shiftUp, it is the distance to the top of the tree, so shiftUp is expensive for nodes at the bottom of the tree
- For shiftDown, it is the distance to the bottom of the tree, so shiftDown is expensive for nodes at the top of the tree

Heapify/BuildHeap Complexity Analysis

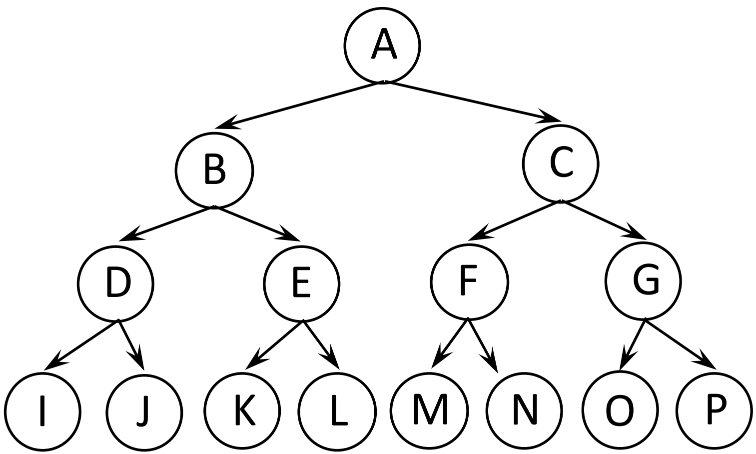
- Although both operations are O(log n) in the worst case, only one node is at the top whereas half the nodes lie in the bottom layer
- So it shouldn't be too surprising that if we have to apply an operation to every node, we would prefer shiftDown over shiftUp



Max # of swaps	Max # of nodes
height h	$1 = 2^0$
height $h - 1$	$2 = 2^1$
height $h - 2$	$4 = 2^2$
height $h - 3$ $= 0$	$8 = 2^3$

Heapify/BuildHeap Complexity Analysis

- Total swaps (max) = $\sum_{i=0}^{h} 2^i(h-i)$
- $A \;=\; h + 2(h-1) + 2^2(h-2) + \cdots + 2^{h-1}1 + 0 \ldots (1)$
- $2A = 2h + 2^2(h-1) + 2^3(h-2) + \cdots + 2^h 1 + 0 \ldots (2)$



Max # of swaps	Max # of nodes
height h	$1 = 2^0$
height $h-1$	$2 = 2^1$
height $h-2$	$4 = 2^2$
height $h-3$ $= 0$	$8 = 2^3$

Heapify/BuildHeap Complexity Analysis

- Total swaps (max) = $\sum_{i=0}^{h} 2^i (h-i)$
- $A \ = \ h + 2(h-1) + 2^2(h-2) + \cdots + 2^{h-1}1 + 0 \ldots (1)$
- $2A = 2h + 2^2(h-1) + 2^3(h-2) + \cdots + 2^h 1 + 0 \ldots (2)$

- (2) – (1) $\Rightarrow A = -h + 2 + 2^2 + 2^3 + \cdots + 2^{h-1} + 2^h$

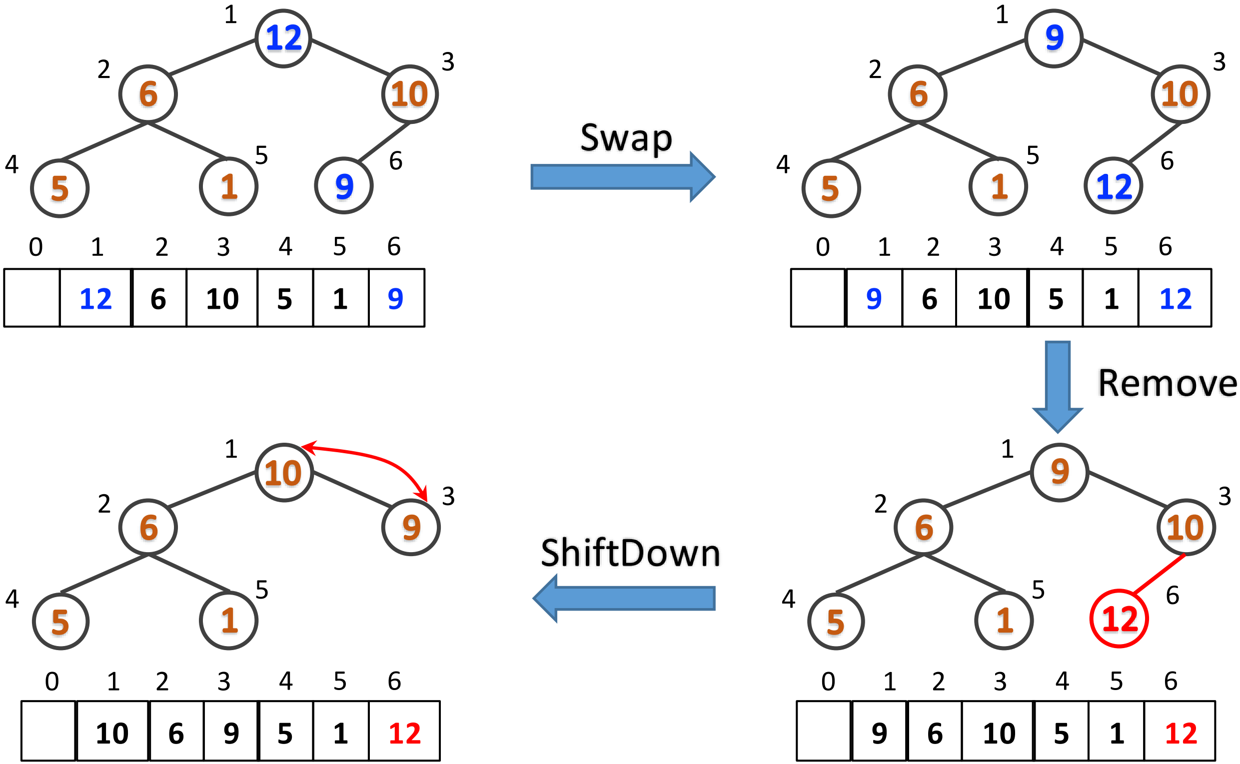
$$= -h + \left(\frac{2^{h+1}-1}{2-1} - 1\right) = (2^{h+1} - 1) - (h+1)$$

$$= (N-1) - (\log N + 1) = O(N)$$

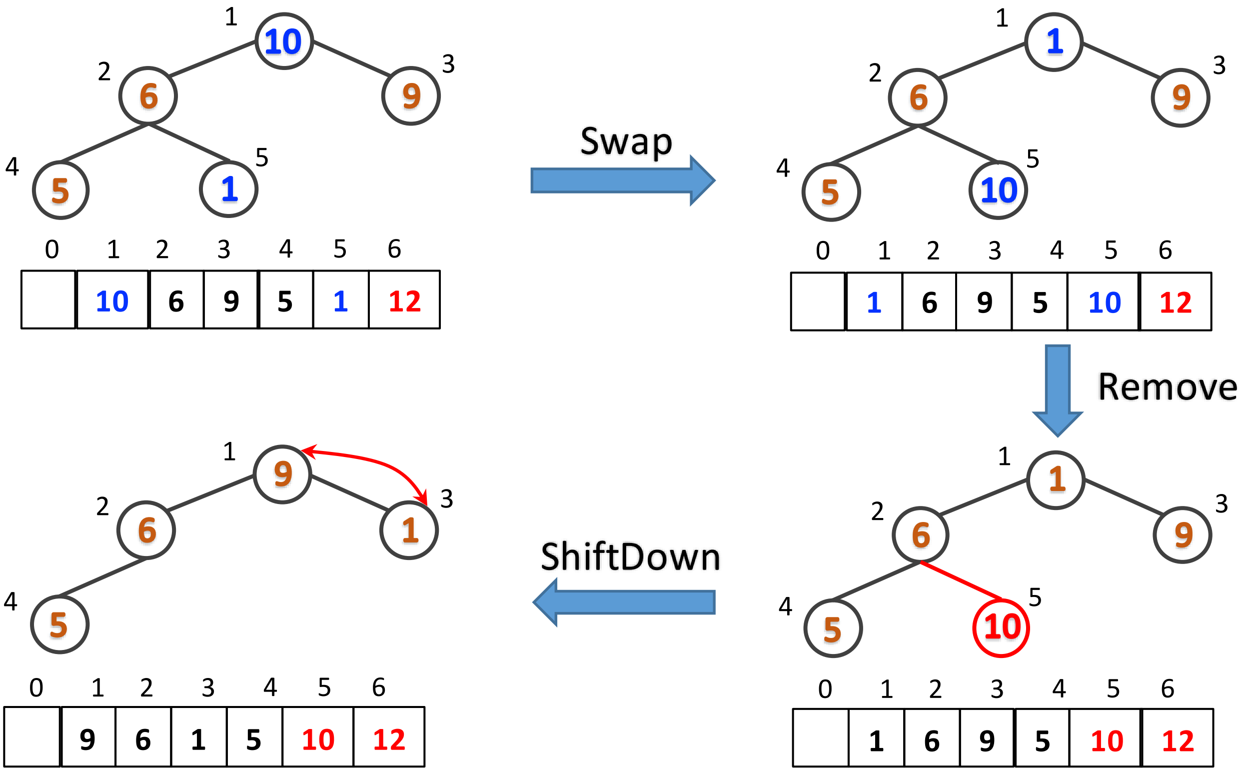
Heapsort

- Heapsort uses heaps to sort in $O(N \log N)$ time
- Basic strategy:
  - Build a min-heap of N elements in $O(N)$ time
  - Perform N pop() operations
  - Store these elements in a second array giving N sorted elements
- Since each pop() takes $O(\log n)$ times, total running time is $O(N \log N)$
- The main problem with this algorithm is that it uses an extra array
- A clever way to avoid this is to use the fact that after each pop() operation, the heap shrinks by 1
- Thus, the previous last cell can be used to store the element that was just popped
- Using this strategy, the array will contain the elements in decreasing sorted order
- If we want in more typical increasing sorted order, we can change the heap to max-heap so that parent has larger element than child

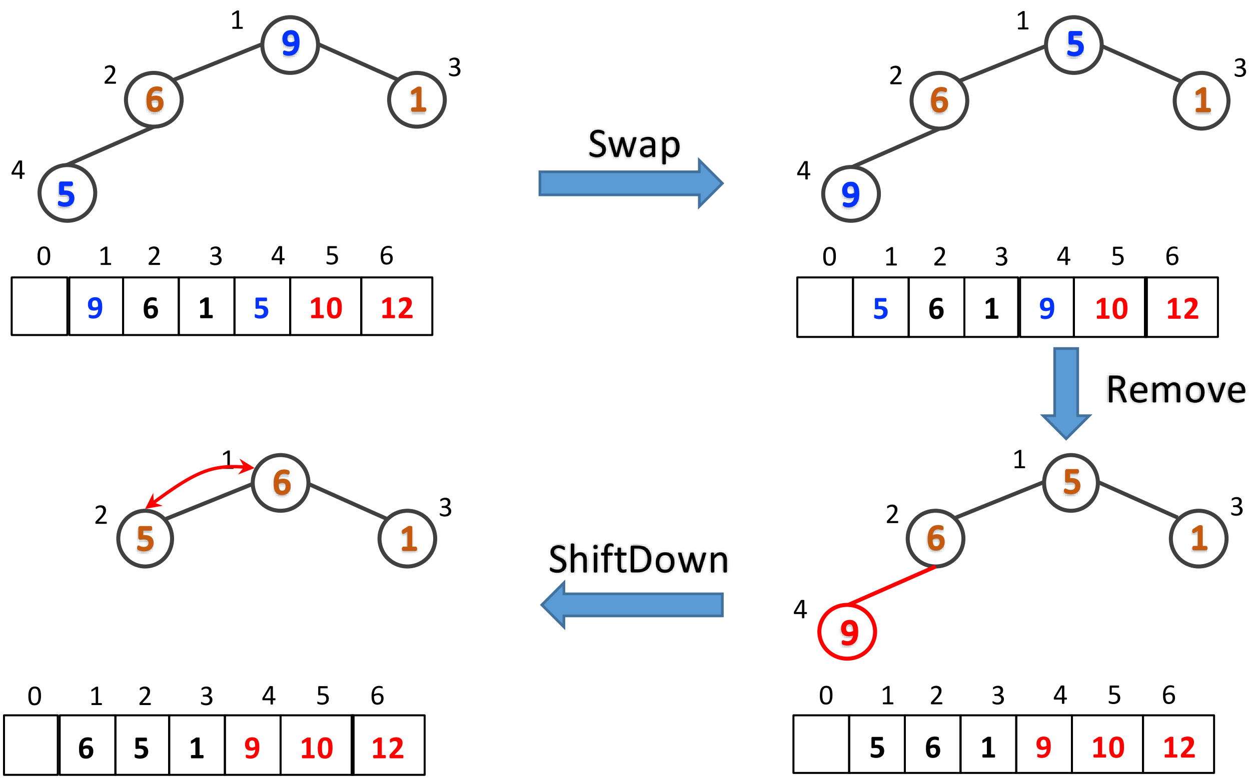
Heapsort



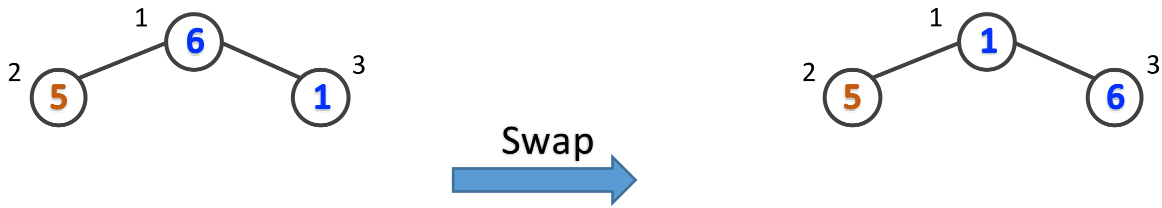
Heapsort


Swap
Remove
ShiftDown
0 1 2 3 4 5 6
10 6 9 5 1 12
1 6 9 5 10 12
1 6 9 5 10 12
9 6 1 5 10 12

Heapsort

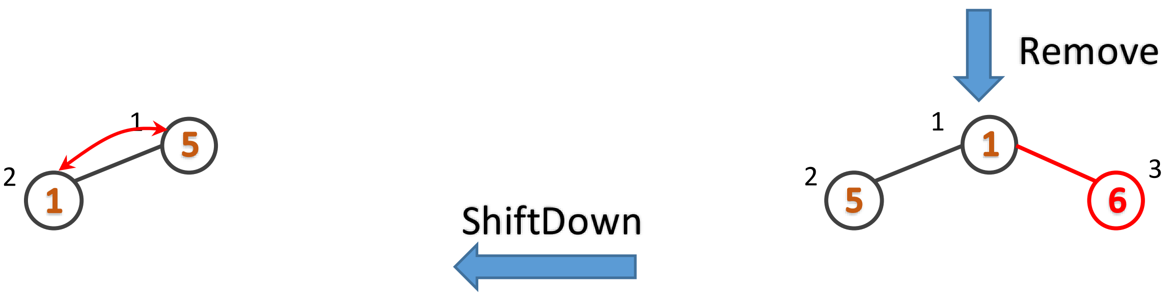

Swap
Remove
ShiftDown
0 1 2 3 4 5 6
9 6 1 5 10 12
5 6 1 9 10 12
5 6 1 9 10 12
6 5 1 9 10 12

Heapsort



0	1	2	3	4	5	6
	6	5	1	9	10	12

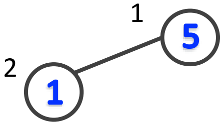
0	1	2	3	4	5	6
	1	5	6	9	10	12



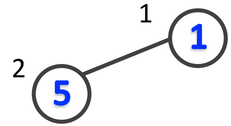
0	1	2	3	4	5	6
	5	1	6	9	10	12

0	1	2	3	4	5	6
	1	5	6	9	10	12

Heapsort



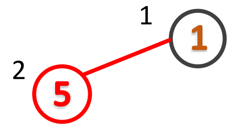
Swap



0	1	2	3	4	5	6
	5	1	6	9	10	12

0	1	2	3	4	5	6
	1	5	6	9	10	12

Remove



0	1	2	3	4	5	6
	1	5	6	9	10	12

0	1	2	3	4	5	6
	1	5	6	9	10	12

Thank You