SOFTWARE SECURITY LABORATORY
HANYANG UNIVERSITY

Simple Data Structures

한양대학교 컴퓨터소프트웨어학부

2017년 2학기

Time Complexity

- How long does it take to do a certain task, in terms of data size? (or, how many operations needed)
  - How fast can you find in a value in a vector?
  - How fast can you find a value in a sorted vector?
  - How fast can you sort a vector?
  - How fast can you insert a value in a vector?
  - How fast can you insert a value in a sorted vector?
  - ...

Big-O Notation

- Big-O notation
    - The rough upper-bound is of most interest.
      - exclude coefficients and low-order terms.
    - For example, $3n^2 + 2n\log n + 3$ is $O(n^2)$.

$O(1)$	constant time	1	1
$O(\log n)$	logarithmic time	2	3
$O(n)$	linear time	100	1,000
$O(n\log n)$	linearithmic time	200	3,000
$O(n^2)$	quadratic time	10,000	1,000,000
$O(n^c)$	polynomial time	100^c	$1{,}000^c$
$O(c^n)$	exponential time	c^{100}	c^{1000}

Vector (Array)

- Consecutively allocated memory space.
- One can randomly access elements by their index.
- Modifying an array by inserting and deleting elements is costly.
- Searching an element in an array is not fast.

1	5	3	15	9	7	13	11

a	zz	abc	boy	##	123

Vector (Array)

- Consecutively allocated memory space.
- One can randomly access elements by their index : O(1).
- Modifying an array by inserting and deleting elements is costly : O(n).
- Searching an element in an array is not fast : O(n).

1	5	3	15	9	7	13	11

a	zz	abc	boy	##	123

```
template <typename T>
class Vector {
 public:
  Vector() : data_(NULL), size_(0), alloc_(0);
  ~Vector() { delete[] data_; }

  void Get(int i) const { return data_[i]; }

  void Insert(int i, const T& v) {
    if (size_ >= alloc_) {
      T* tmp = new T[alloc_ = size_ + 1];
      for (int k = 0; k < size_; ++k) tmp[k] = data_[k];
      delete[] data_;
      data_ = tmp;
    }
    for (int k = size_; k > i; ++k) data_[k] = data_[k - 1];
    data_[i] = v;
    ++size_;
  }

  void Erase(int i) {
    for (int k = i + 1; k < size_; ++k) data_[k - 1] = data_[k];
    --size_;
  }

  int Find(const T& v) {
    for (int k = 0; k < size_; ++k) if (data_[k] == v) return k;
    return -1;
  }

 private:
  T* data_;
  int size_, alloc_;
};
```

Sorted Array

- Consecutively allocated memory space storing values sorted.
- Searching an element in an array is efficient : O(log$n$).
- Inserting and deleting elements is costly : O(n).

1	3	5	7	9	11	13	15

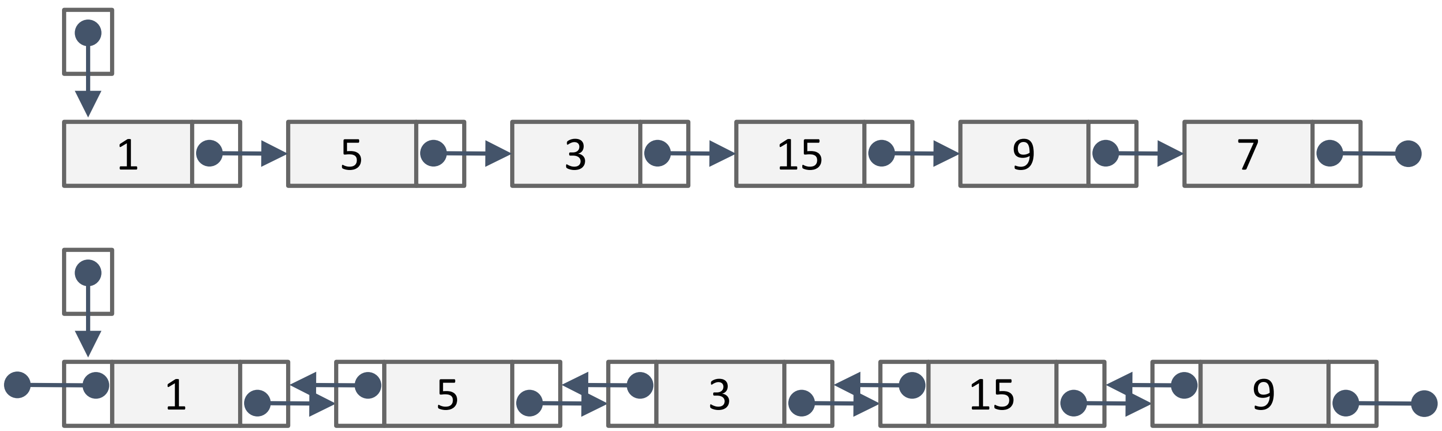
123	a	abc	boy	def	zz

```cpp
template <typename T>
class SortedArray {
 public:
  SortedArray() : data_(0), size_(0), alloc_(0) {}
  ~SortedArray() { delete[] data_; }

  void Insert(const T& v);
  void Erase(const T& v);

  int Find(const T& v) { return Find(v, NULL); }

  int size() const { return size_; }
  int alloc() const { return alloc_; }
  const T& operator[](int i) const { return data_[i]; }

 private:
  int Find(const T& v, int* idx) {
    int i = 0, j = size_, k = size_ / 2;
    while (i < j) {
      k = (i + j) / 2;
      if (data_[k] == v) break;
      if (data_[k] > v) j = k;
      else i = k + 1;
    }
    if (idx) *idx = i;
    return (i < j) ? k : -1;
  }

  T* data_;
  int size_, alloc_;
};
```

```
template <typename T>
class SortedArray {
 public:
  ...

  void Insert(const T& v) {
    int idx = -1;
    if (Find(v, &idx) >= 0) return;
    if (size_ >= alloc_) {
      T* tmp = new T[alloc_ = size_ + 1];
      for (int k = 0; k < size_; ++k) tmp[k] = data_[k];
      delete[] data_;
      data_ = tmp;
    }
    for (int k = size_; k > idx; --k) data_[k] = data_[k - 1];
    data_[idx] = v;
    ++size_;
  }

  void Erase(const T& v) {
    int idx = -1;
    if (Find(v, &idx) < 0) return;
    for (int k = idx + 1; k < size_; ++k) data_[k - 1] = data_[k];
    --size_;
  }

  ...
};
```

Linked List

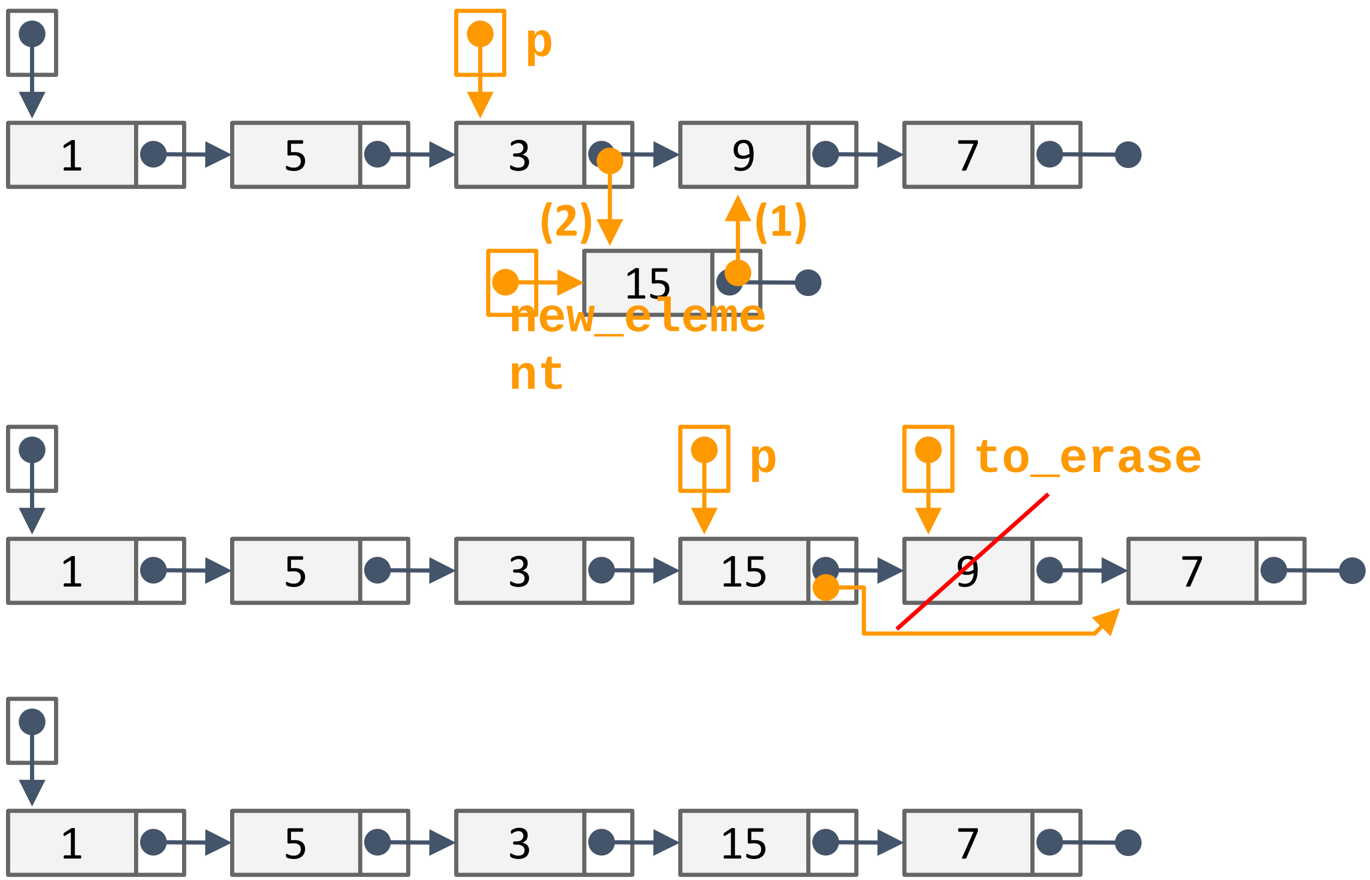
- Linked list of cells storing elements.
- Singly and doubly linked lists.
- Random access by an index is costly : O(n).
- Inserting and deleting random elements is also costly : O(n).
- Inserting and deleting elements at both ends is efficient : O(1).
- Searching an element in a list is not fast : O(n).



```
template <typename T>
class SinglyLinkedList {
 public:
  struct Element { T v; Element* next; };

  SinglyLinkedList() : head_(NULL) {}

  ~SinglyLinkedList() {
    for (const Element *p = head_, *q = NULL; p != NULL; p = q) {
      q = p->next;
      delete p;
    }
  }

  int Size() const {
    int n = 0;
    for (const Element* p = head_; p != NULL; p = p->next) ++n;
    return n;
  }

  const T* Get(int i) const {
    Element* p = head_;
    for (int k = 0; p != NULL && k < i; ++k) p = p->next;
    return p != NULL ? &p->v : NULL;
  }

  void Insert(int i, const T& v);
  void Erase(int i);

 private:
  Element* head_;
};
```

Linked List

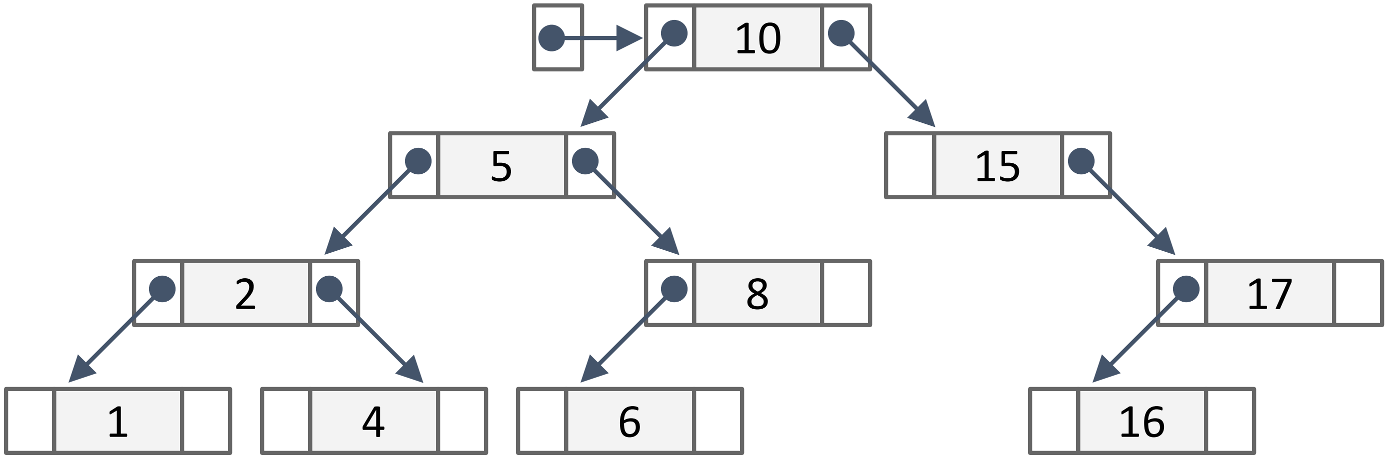
- Inserting and deleting elements is not simple.



```
template <typename T>
void SinglyLinkedList<T>::Insert(int i, const T& v) {
  Element* p = head_;
  for (int k = 0; p != NULL && k < i - 1; ++k) p = p->next;
  Element* new_element = new Element;
  new_element->v = v;
  if (i <= 0) {
    new_element->next = head_;
    head_ = new_element;
  } else {
    new_element->next = p->next;
    p->next = new_element;
  }
}

template <typename T>
void SinglyLinkedList<T>::Erase(int i) {
  if (head_ == NULL) return;
  Element* p = head_;
  Element* to_erase = NULL;
  for (int k = 0; k < i - 1 && p != NULL; ++k) p = p->next;
  if (i <= 0) {
    to_erase = head_;
    head_ = head_->next;
  } else {
    to_erase = p->next;
    if (p->next != NULL) p->next = p->next->next;
  }
  delete to_erase;
}
```

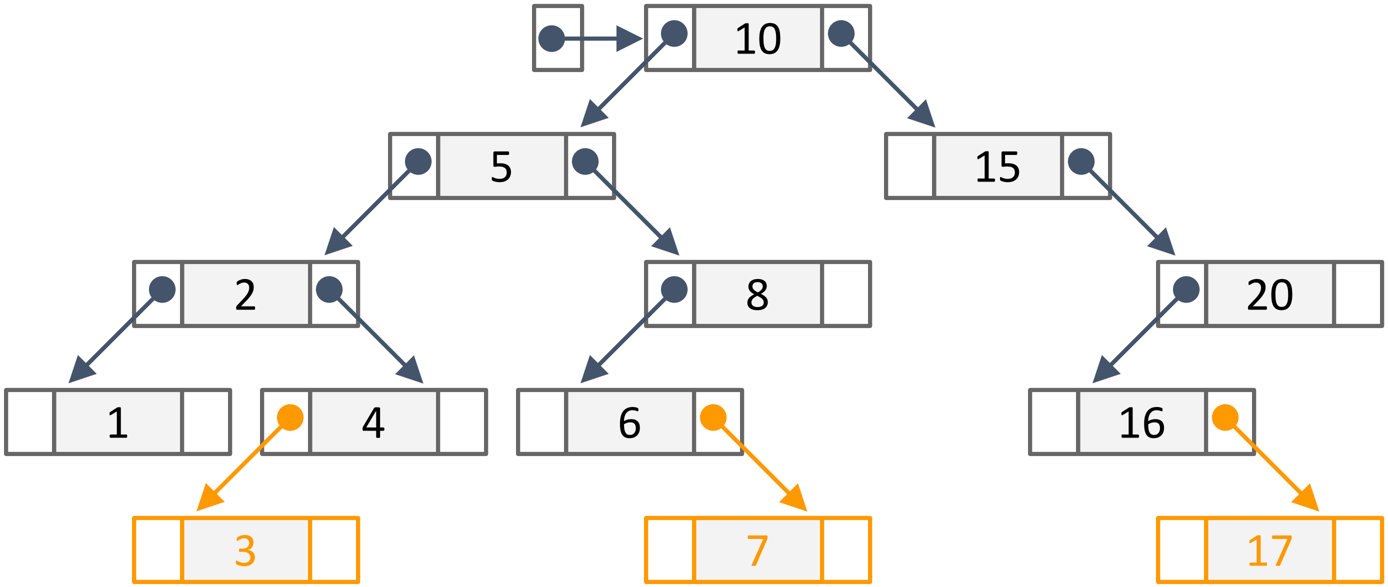
Binary Search Tree

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes.



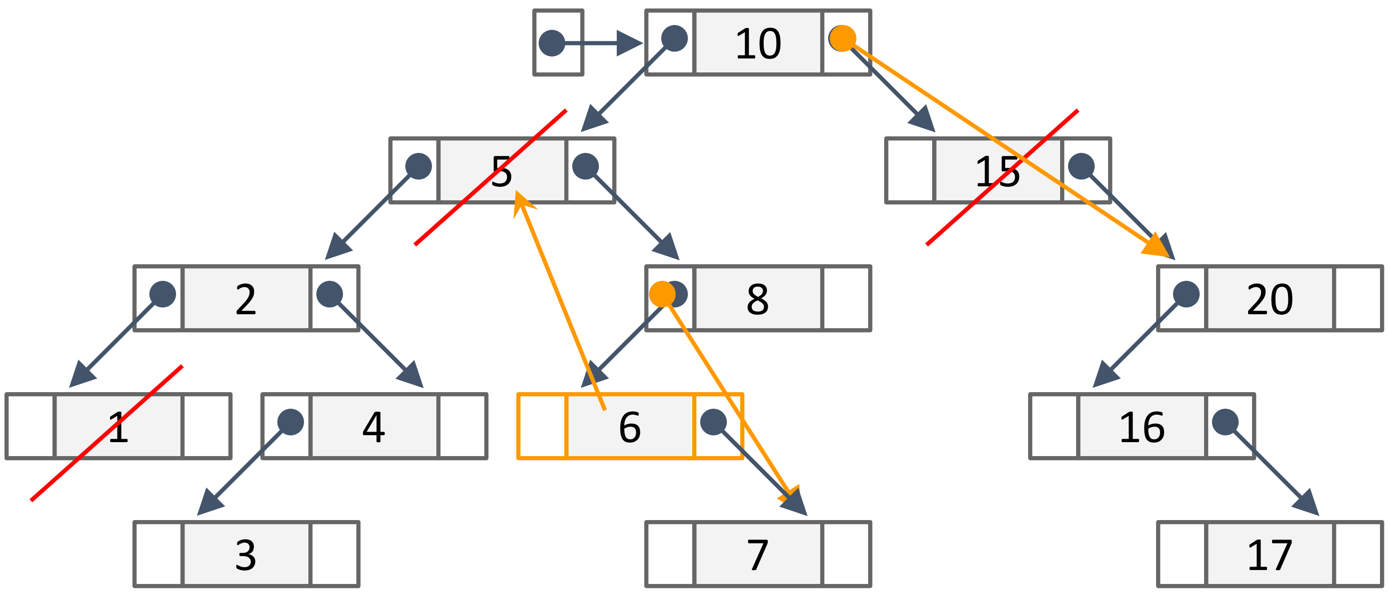
Binary Search Tree

	Average	Worst
Search	O(log$n$)	O(n)
Insert	O(log$n$)	O(n)
Delete	O(log$n$)	O(n)



Binary Search Tree

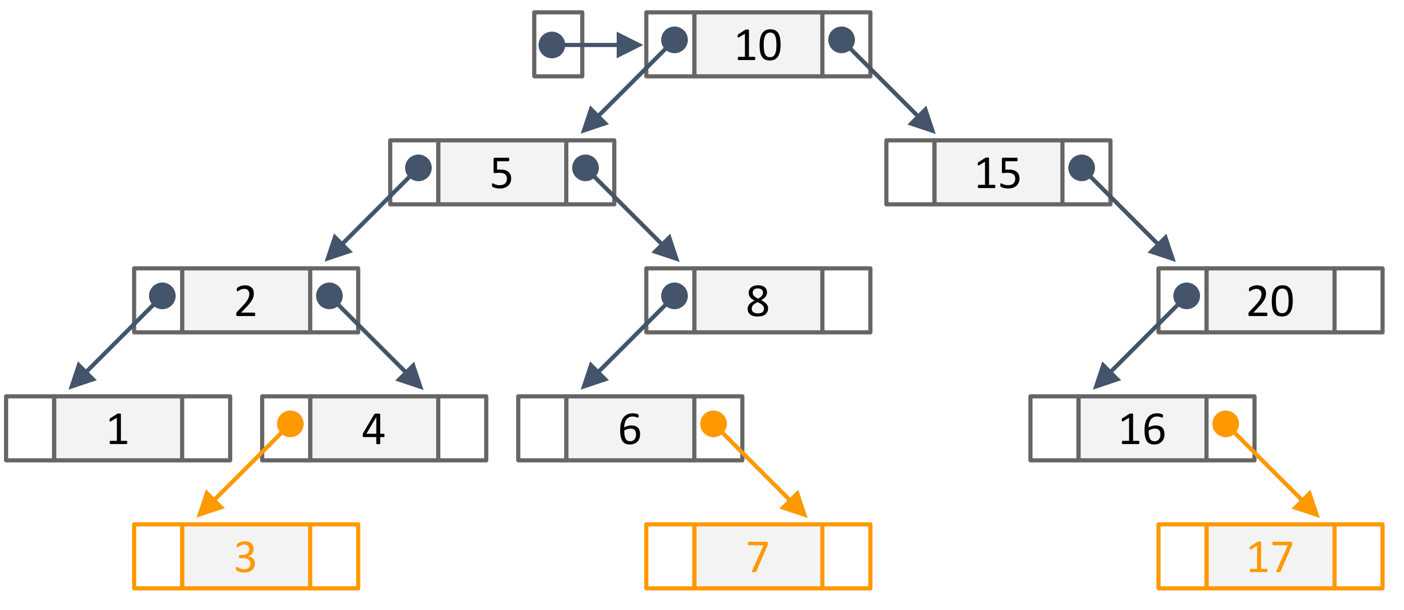
- Delete : if the node to be deleted has both children,
  - Find the smallest node in the right subtree (or largest in the left).
  - Replace it with the node and delete it.



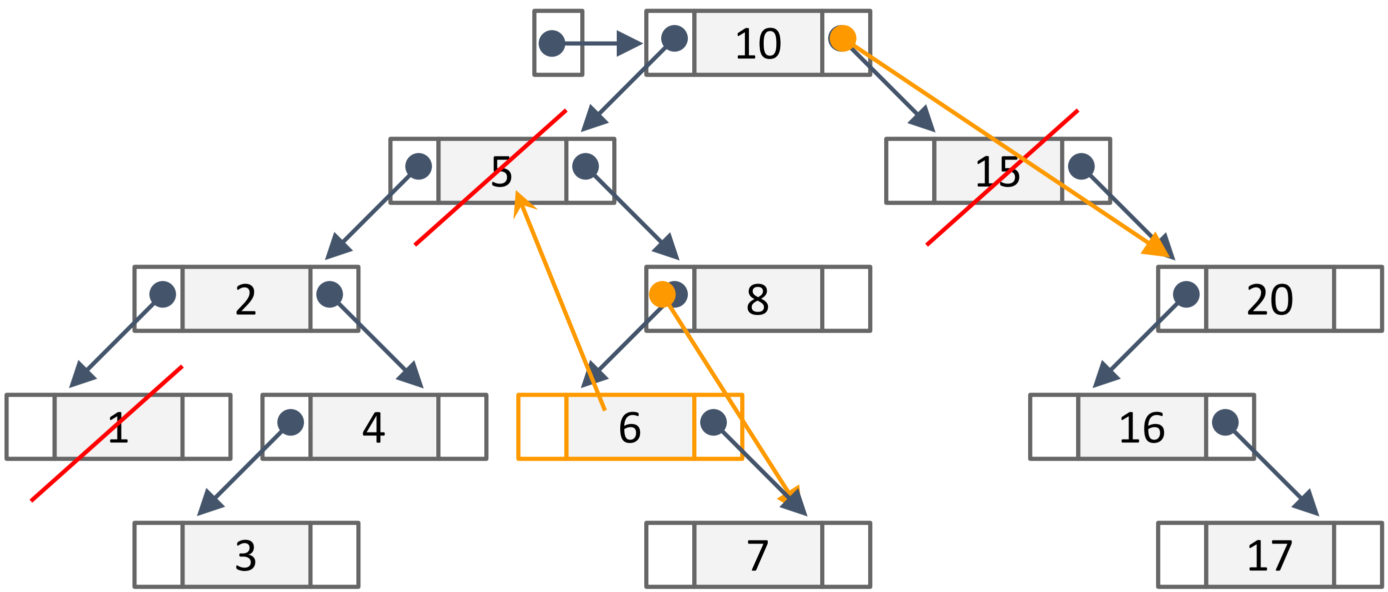
```
template <typename T>
class BinarySearchTree {
 public:

  BinarySearchTree() : root_(NULL) {}
  ~BinarySearchTree() { if (root_) root_->DeleteSubtree(); }

  int Size() const { return root_ ? root_->Size() : 0; }
  const T* Find(const T& v) const { return root_ ? root_->Find(v) : NULL; }

  void Insert(const T& v);
  void Erase(const T& v);

  void Dump() { if (root_) root_->DumpSubtree(); };

 private:
  struct Node {
    T val;
    Node* left;
    Node* right;
  };

  Node* root_;
};
```

```
template <typename T>
class BinarySearchTree {
  ...
 private:
  struct Node {
    Node(const T& v) : val(v), left(NULL), right(NULL) {}

    void DeleteSubtree() {
      if (left != NULL) left->DeleteSubtree();
      if (right != NULL) right->DeleteSubtree();
      delete this;
    }
    void DumpSubtree() {
      cout << "(";
      if (left != NULL) left->DumpSubtree();
      cout << " " << val << " ";
      if (right != NULL) right->DumpSubtree();
      cout << ")";
    }

    int Size() const {
      return 1 + (left ? left->Size() : 0) + (right ? right->Size() : 0);
    }

    const T* Find(const T& v) const {
      if (val == v) return &val;
      return (v < val) ? (left ? left->Find(v) : NULL) :
                         (right ? right->Find(v) : NULL);
    }

    T val;
    Node* left;
    Node* right;
  };

  Node* root_;
};
```

```
template <typename T>
void BinarySearchTree<T>::Insert(const T& v) {
  if (root_ == NULL) {
    root_ = new Node(v);
  } else {
    Node* p = root_;
    while (p != NULL) {
      if (v == p->val ||
          (v < p->val && p->left == NULL) ||
          (v > p->val && p->right == NULL)) break;
      p = (v < p->val) ? p->left : p->right;
    }
    if (v < p->val) p->left = new Node(v);
    else if (v > p->val) p->right = new Node(v);
  }
}
```

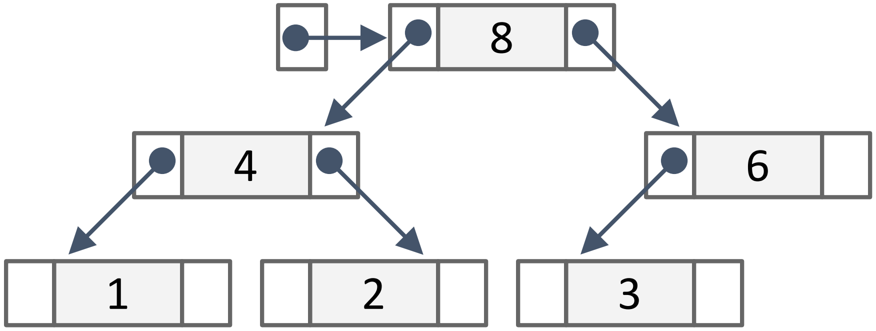


```
template <typename T>
void BinarySearchTree<T>::Erase(const T& v) {
  if (root_ == NULL) return;
  Node** pp = &root_;
  while (*pp != NULL) {
    if (v == (*pp)->val) break;
    pp = &(v < (*pp)->val ? (*pp)->left : (*pp)->right);
  }
  if (*pp == NULL) return;
  Node* p = *pp;
  if (p->left != NULL && p->right != NULL) {
    Node** pq = &(p->right);
    while ((*pq)->left != NULL) pq = &(*pq)->left;
    p = *pq;
    *pq = p->right;
    (*pp)->val = p->val;
  } else {
    *pp = (p->left ? p->left : p->right)
  }
  delete p;
}
```



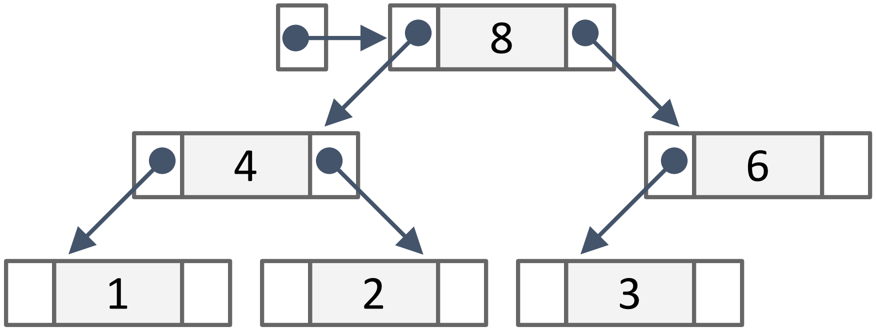
Heap

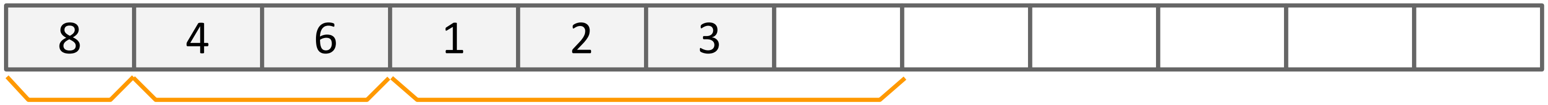
- Heap is a specialized tree-based data structure.
- Max heap : the keys of parent nodes are always greater than or equal to those of the children and the highest key is in the root node.
    - heapify: create a heap out of given array of elements.
    - find-max: find the maximum item of a max-heap.
    - delete-max: removing the root node of a max-heap.
    - increase-key: updating a key within a max-heap.
    - insert: adding a new key to the heap.



Heap Representation

- A binary heap can be represented as an array.
    - Parent of node i is floor(i/2). (all indices are one-based.)
    - Children of node i is (2*i) and (2*i+1).





```cpp
template <typename T>
class MaxHeap {
 public:
  MaxHeap() {}

  void Set(const vector<T>& vec) {
    vec_ = vec;
    int last_parent_idx = ParentIndex(vec_.size() - 1);
    for (int i = last_parent_idx; i >= 0; --i) Heapify(i);
  }

  int Size() const { return vec_.size(); }
  const T& GetMax() const { return vec_.front(); }

 private:
  void Heapify(int parent) {
    int i = ChildIndex(parent);
    if (i < vec_.size() && vec_[parent] < vec_[i]) {
      swap(vec_[parent], vec_[i]);
      Heapify(i);
    }
    if (i + 1 < vec_.size() && vec_[parent] < vec_[i + 1]) {
      swap(vec_[parent], vec_[i + 1]);
      Heapify(i + 1);
    }
  }
  static int ParentIndex(int i) { return (i + 1) / 2 - 1; }
  static int ChildIndex(int i) { return 2 * i + 1; }

  vector<T> vec_;
};
```

```
template <typename T>
class MaxHeap {
 public:
  MaxHeap() {}

  void DeleteMax() {
    swap(vec_.front(), vec_.back());
    vec_.pop_back();
    Heapify(0);
  }

  void Insert(const T& v) {
    vec_.push_back(v);
    for (int i = ParentIndex(vec_.size() - 1); i >= 0; i = ParentIndex(i)) {
      Heapify(i);
    }
  }

 private:
  void Heapify(int parent) {
    int i = ChildIndex(parent);
    if (i < vec_.size() && vec_[parent] < vec_[i]) {
      swap(vec_[parent], vec_[i]);
      Heapify(i);
    }
    if (i + 1 < vec_.size() && vec_[parent] < vec_[i + 1]) {
      swap(vec_[parent], vec_[i + 1]);
      Heapify(i + 1);
    }
  }
  static int ParentIndex(int i) { return (i + 1) / 2 - 1; }
  static int ChildIndex(int i) { return 2 * i + 1; }

  vector<T> vec_;
};
```


Q&A