Lecture7: 우선순위 큐(8.1절), 힙(8.3절)

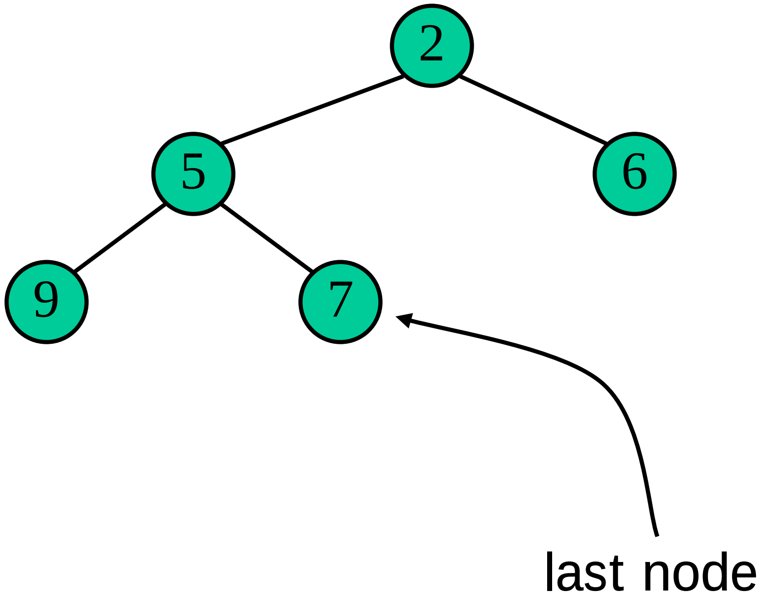
김 강 희
khhkim@ssu.ac.kr

요약

- 우선순위 큐
  - (key, value)로 구성되는 엔트리들을 저장하는 자료 구조로서 key는 우선순위를 나타낸다.
  - 승객 대기자, 경매, 주식 시장 등 우선순위 개념이 적용되는 영역에서 필요하다.
  - 우선순위 큐 정렬의 계산 복잡도는 O(N×logN)으로서 O(N^2)인 선택 정렬(selection sort) 또는 삽입 정렬(insertion sort)보다 월등히 우수하다.

요약

- 힙(heap)
  - 다음 성질을 갖는 이진 트리를 말한다.
    - 힙 오더(heap order): 루트 아닌 모든 내부 노드 v에 대해 $key(v) \geq key(parent(v))$
  - 효율성을 위해서 완전 이진 트리(complete binary tree) 특성을 갖는 것이 좋다.
    - 마지막 레벨을 제외하고 모든 노드가 채워진 이진 트리. 마지막 레벨의 노드들은 왼쪽으로 채워져 있다. 마지막 레벨이 다 채워질 수도 있다.



8.1 Priority Queue ADT

```
template <typename E, typename C>
 class PriorityQueue {
 public:
   int size() const;
   bool isEmpty() const;
   void insert(const E& e);
   const E& min() const throw(QueueEmpty);
   void removeMin() throw(QueueEmpty);
 };
```

Priority Queue 사용 예시

연산	출력	우선순위 큐
insert(5)	–	{5}
insert(9)	–	{5,9}
insert(2)	–	{2,5,9}
insert(7)	–	{2,5,7,9}
min()	[2]	{2,5,7,9}
removeMin()	–	{5,7,9}
size()	3	{5,7,9}
min()	[5]	{5,7,9}
removeMin()	–	{7,9}
removeMin()	–	{9}
removeMin()	–	{}
empty()	*true*	{}
removeMin()	"error"	{}

Total Order Relations

- Keys in a priority queue can be arbitrary objects on which an order is defined
- Two distinct entries in a priority queue can have the same key
- Mathematical concept of total order relation $\leq$
  - Reflexive property: $x \leq x$
  - Antisymmetric property: $x \leq y \wedge y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Comparator ADT

- Implements the boolean function isLess(p,q), which tests whether p < q
- Can derive other relations from this:
  - (p == q) is equivalent to
  - (!isLess(p, q) && !isLess(q, p))
- Can implement in C++ by overloading "()"

Two ways to compare 2D points:

```
class LeftRight { // left-right comparator
public:
    bool operator()(const Point2D& p,
            const Point2D& q) const
    { return p.getX() < q.getX(); }
};
class BottomTop { // bottom-top
public:
    bool operator()(const Point2D& p,
    const Point2D& q) const
    { return p.getY() < q.getY(); }
};
```

Priority Queue Sorting

- We can use a priority queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of insert operations
  2. Remove the elements in sorted order with a series of removeMin operations
- The running time of this sorting method depends on the priority queue implementation

Algorithm ***PQ-Sort***(***S***, ***C***)
    Input sequence ***S***, comparator ***C*** for the elements of ***S***
    Output sequence ***S*** sorted in increasing order according to ***C***
    P ← priority queue with
        comparator ***C***
    while ¬***S.empty*** ()
        e ← ***S.front***(); ***S.eraseFront***()
        P.insert (***e***, ∅)
    while ¬***P.empty***()
        e ← ***P.removeMin***()
        S.insertBack(***e***)

Sequence-based Priority Queue

- Implementation with an unsorted list

- Performance:
  - insert takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - removeMin and min take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

- Implementation with a sorted list

- Performance:
  - insert takes $O(n)$ time since we have to find the place where to insert the item
  - removeMin and min take $O(1)$ time, since the smallest key is at the beginning

Selection-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence
- Running time of Selection-sort:
  1. Inserting the elements into the priority queue with n insert operations takes $O(n)$ time
  2. Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to

$$1 + 2 + \ldots + n$$

- Selection-sort runs in $O(n^2)$ time

Selection-Sort Example

	Sequence S	Priority Queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(7,4)
..	..	..
(g)	()	(7,4,8,2,5,3,9)
Phase 2		
(a)	(2)	(7,4,8,5,3,9)
(b)	(2,3)	(7,4,8,5,9)
(c)	(2,3,4)	(7,8,5,9)
(d)	(2,3,4,5)	(7,8,9)
(e)	(2,3,4,5,7)	(8,9)
(f)	(2,3,4,5,7,8)	(9)
(g)	(2,3,4,5,7,8,9)	()

Insertion-Sort

- Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence
- Running time of Insertion-sort:
  1. Inserting the elements into the priority queue with n insert operations takes time proportional to

     $$1 + 2 + \ldots + n$$

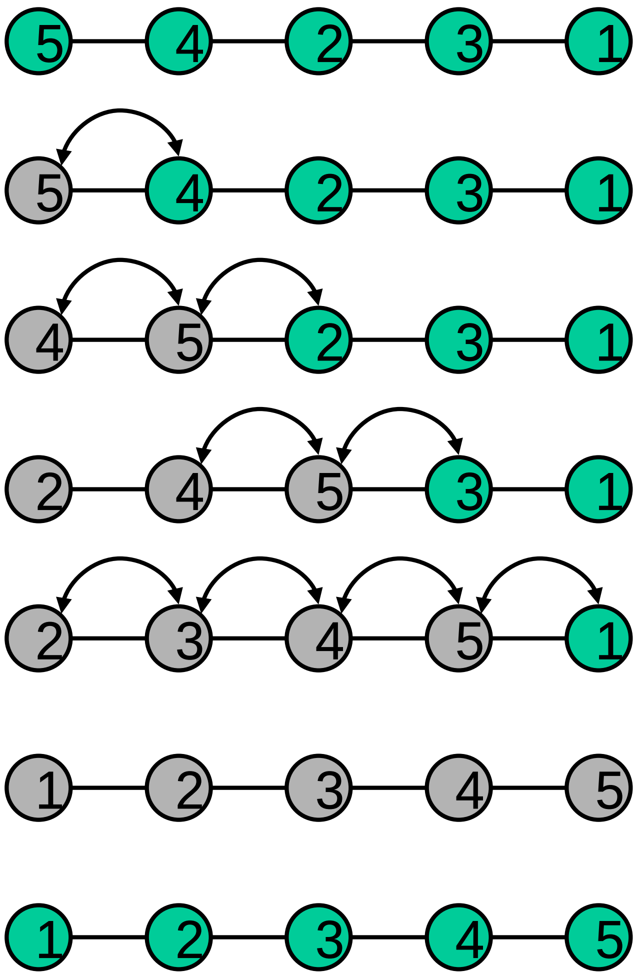
  2. Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time
- Insertion-sort runs in $O(n^2)$ time

Insertion-Sort Example

	Sequence S	Priority queue P
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
(g)	(2,3,4,5,7,8,9)	()

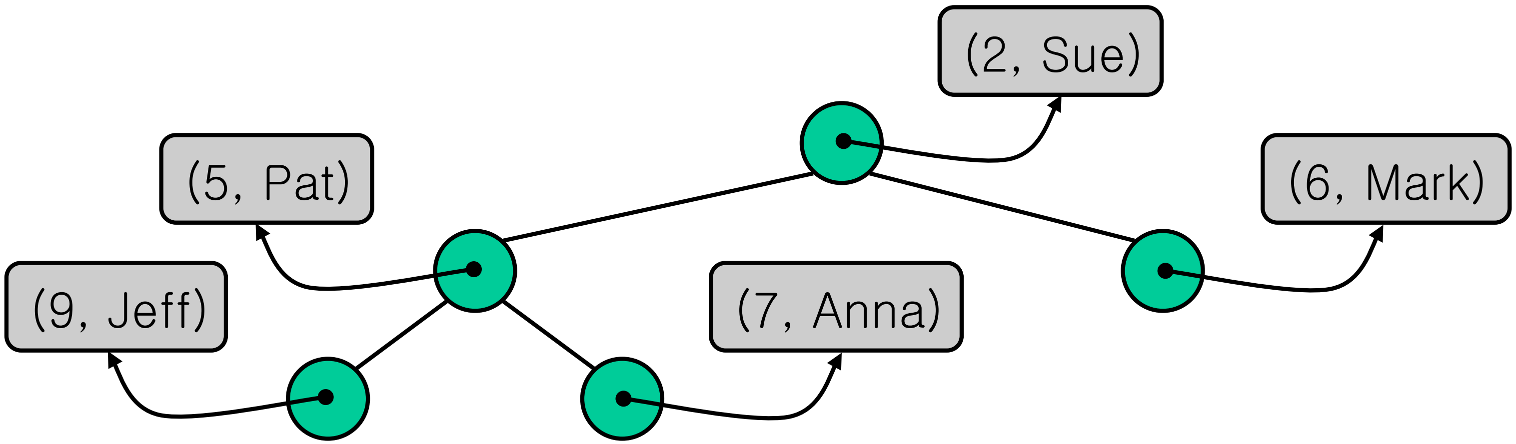
In-place Insertion-Sort

- Instead of using an external data structure, we can impl ement selection-sort and in sertion-sort in-place
- A portion of the input sequ ence itself serves as the pri ority queue
- For in-place insertion-sort
  - We keep sorted the initial portion of the sequence
  - We can use swaps instea d of modifying the seque nce



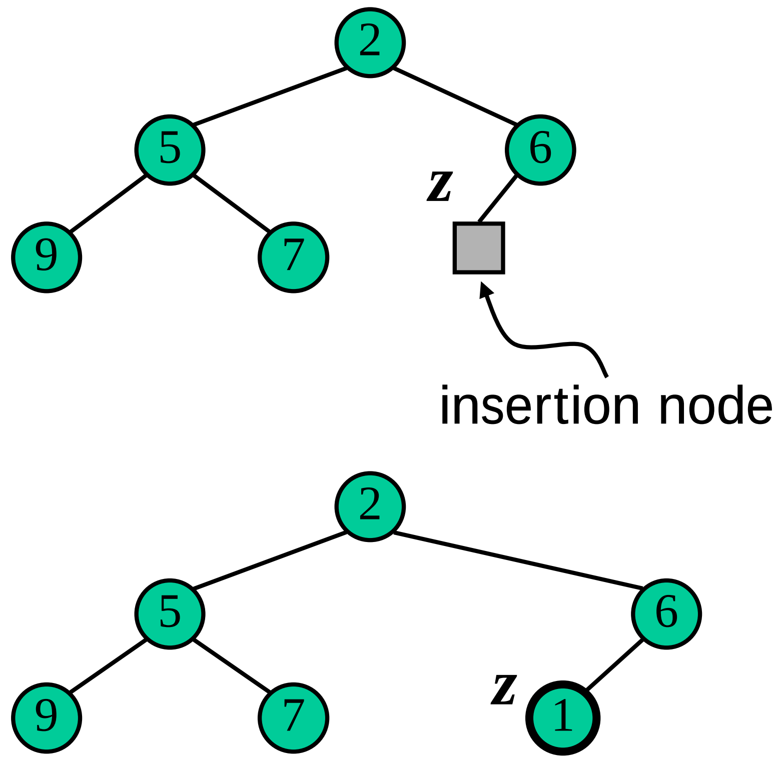
8.3 Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node



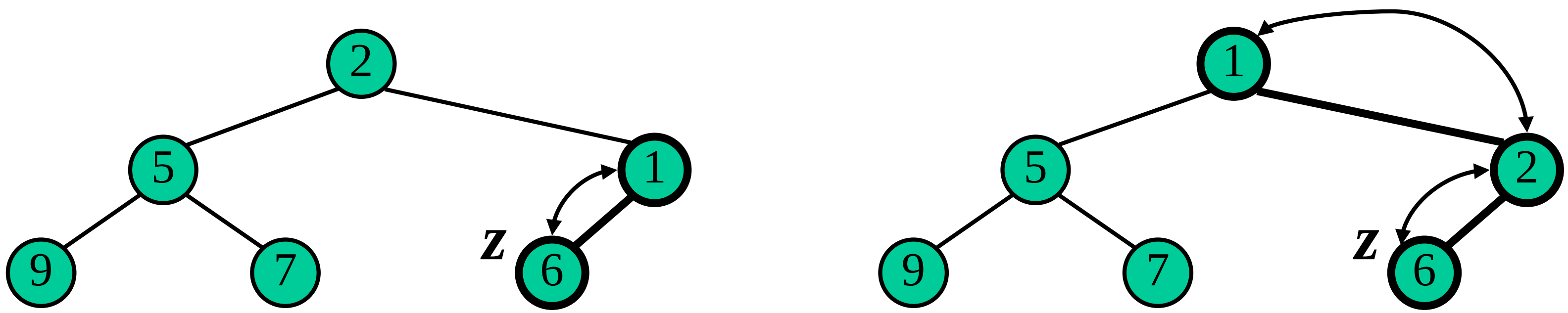
Insertion into a Heap

- Method insertItem of the priority queue ADT corresponds to the insertion of a key k to the heap
- The insertion algorithm consists of three steps
  - Find the insertion node z (the new last node)
  - Store k at z
  - Restore the heap-order property (discussed next)



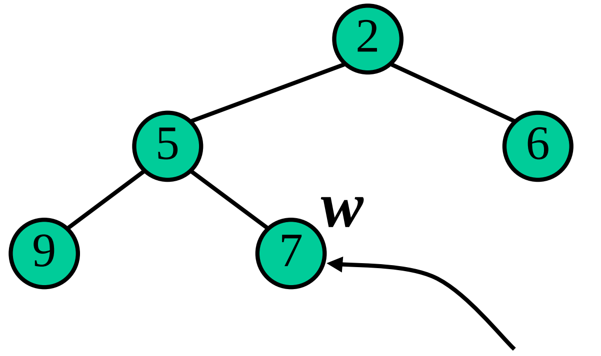
Upheap

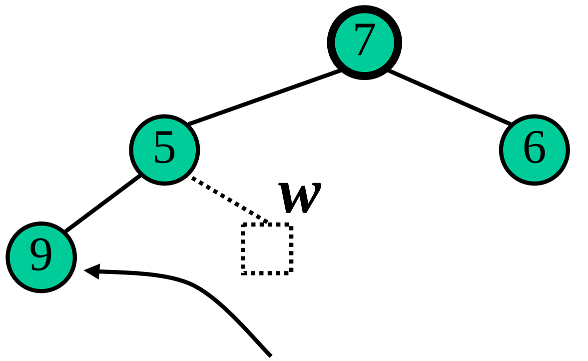
- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time



Removal from a Heap

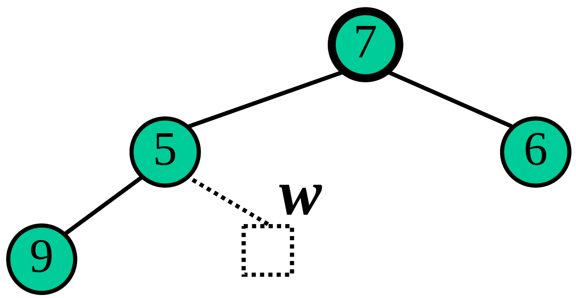
- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node ***w***
  - Remove ***w***
  - Restore the heap-order property (discussed next)

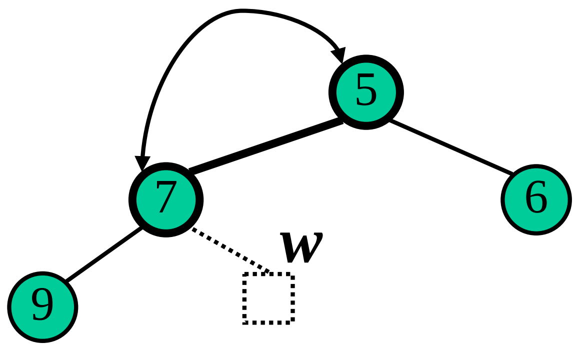




Downheap

- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root
- Upheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time





Heapsort

- 힙 자료구조로 구현된 n개 노드를 가진 우선순위 큐를 고려하자.
- 공간복잡도: O(n)
- insert/removeMin 함수의 시간복잡도: O(log n)
- 힙 기반 우선순위 큐를 사용하면, n개 원소들을 O(n log n) 시간에 정렬할 수 있다.

TopDownHeapSort 함수 예시

```
void TopDownHeapSort(list<Point2D> &pl) {
  Point2D p;
  int n = pl.size();
  HeapPriorityQueue<Point2D, LeftRight> T;

  if (pl.empty())
    return;

  list<Point2D>::iterator it;
  for (it = pl.begin(); it != pl.end(); ++it)
    T.insert(*it);

  pl.clear();
  for (int i = 0; i < n; i++) {
    p = T.min();
    pl.push_back(p);
    T.removeMin();
  }
  return;
}
```

HeapPriorityQueue

```
template <typename E, typename C>
class HeapPriorityQueue {
public:
  int size() const;              // number of elements
  bool empty() const;            // is the queue empty?
  void insert(const E& e);       // insert element
  const E& min();                // minimum element
  void removeMin();              // remove minimum
private:
  VectorCompleteTree<E> T;       // priority queue contents
  C isLess;                      // less-than comparator
                                 // shortcut for tree position
  typedef typename VectorCompleteTree<E>::Position Position;
};
```

HeapPriorityQueue

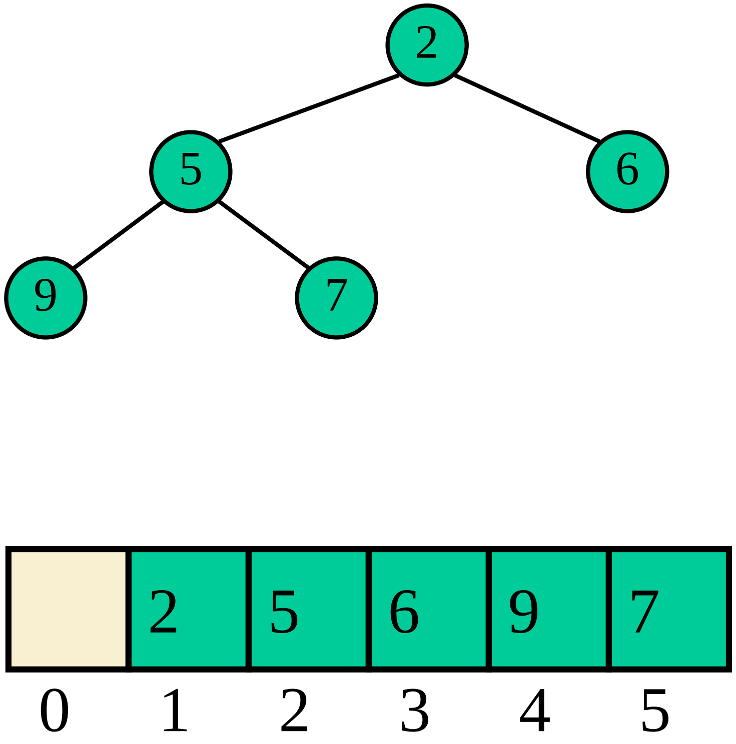
```
template <typename E, typename C>       // number of elements
int HeapPriorityQueue<E,C>::size() const { return T.size(); }

template <typename E, typename C>       // is the queue empty?
bool HeapPriorityQueue<E,C>::empty() const { return size() == 0; }

template <typename E, typename C>       // minimum element
const E& HeapPriorityQueue<E,C>::min()  { return *(T.root()); }

template <typename E, typename C>       // insert element
void HeapPriorityQueue<E,C>::insert(const E& e) {
  T.addLast(e);                   // add e to heap
  Position v = T.last();          // e's position
  while (!T.isRoot(v)) {          // up-heap bubbling
    Position u = T.parent(v);
    if (!isLess(*v, *u)) break;   // if v in order, we're done
    T.swap(v, u);                 // ...else swap with parent
    v = u;
  }
}
```

HeapPriorityQueue

```
template <typename E, typename C>       // remove minimum
void HeapPriorityQueue<E,C>::removeMin() {
  if (size() == 1)              // only one node?
    T.removeLast();             // ...remove it
  else {
    Position u = T.root();        // root position
    T.swap(u, T.last());          // swap last with root
    T.removeLast();               // ...and remove last
    while (T.hasLeft(u)) {        // down-heap bubbling
      Position v = T.left(u);
      if (T.hasRight(u) && isLess(*(T.right(u)), *v))
    v = T.right(u);               // v is u's smaller child
      if (isLess(*v, *u)) {       // is u out of order?
    T.swap(u, v);                 // ...then swap
    u = v;
      }
      else break;                 // else we're done
    }
  }
}
```

Vector-based Heap Implementation

- We can represent a heap with n keys by means of a vector of length $n+1$
- For the node at rank i
  - the left child is at rank $2i$
  - the right child is at rank $2i+1$
- Links between nodes are not explicitly stored
- The cell of at rank 0 is not used
- Operation insert corresponds to inserting at rank $n+1$
- Operation removeMin corresponds to removing at rank n
- Yields in-place heap-sort



VectorCompleteTree

```
template <typename E>
class VectorCompleteTree {
private:                    // member data
  std::vector<E> V;         // tree contents
public:                     // publicly accessible types
  typedef typename std::vector<E>::iterator Position; // a position in the tree
protected:                  // protected utility functions
  Position pos(int i)       // map an index to a position
  { return V.begin() + i; }
  int idx(const Position& p) const      // map a position to an index
  { return p - V.begin(); }
public:
  …
};
```

VectorCompleteTree

```
template <typename E>
class VectorCompleteTree {
  ...
public:
  VectorCompleteTree() : V(1) {}      // constructor
  int size() const           { return V.size() - 1; }
  Position left(const Position& p)       { return pos(2*idx(p)); }
  Position right(const Position& p)     { return pos(2*idx(p) + 1); }
  Position parent(const Position& p)      { return pos(idx(p)/2); }
  bool hasLeft(const Position& p) const{ return 2*idx(p) <= size(); }
  bool hasRight(const Position& p) const { return 2*idx(p) + 1 <= size(); }
  bool isRoot(const Position& p) const { return idx(p) == 1; }
  Position root()            { return pos(1); }
  Position last()            { return pos(size()); }
  void addLast(const E& e)         { V.push_back(e); }
  void removeLast()             { V.pop_back(); }
  void swap(const Position& p, const Position& q)
  { E e = *q; *q = *p; *p = e; }
};
```

감사합니다!