# 자료구조(Data Structure)

# Programming Assignment 5

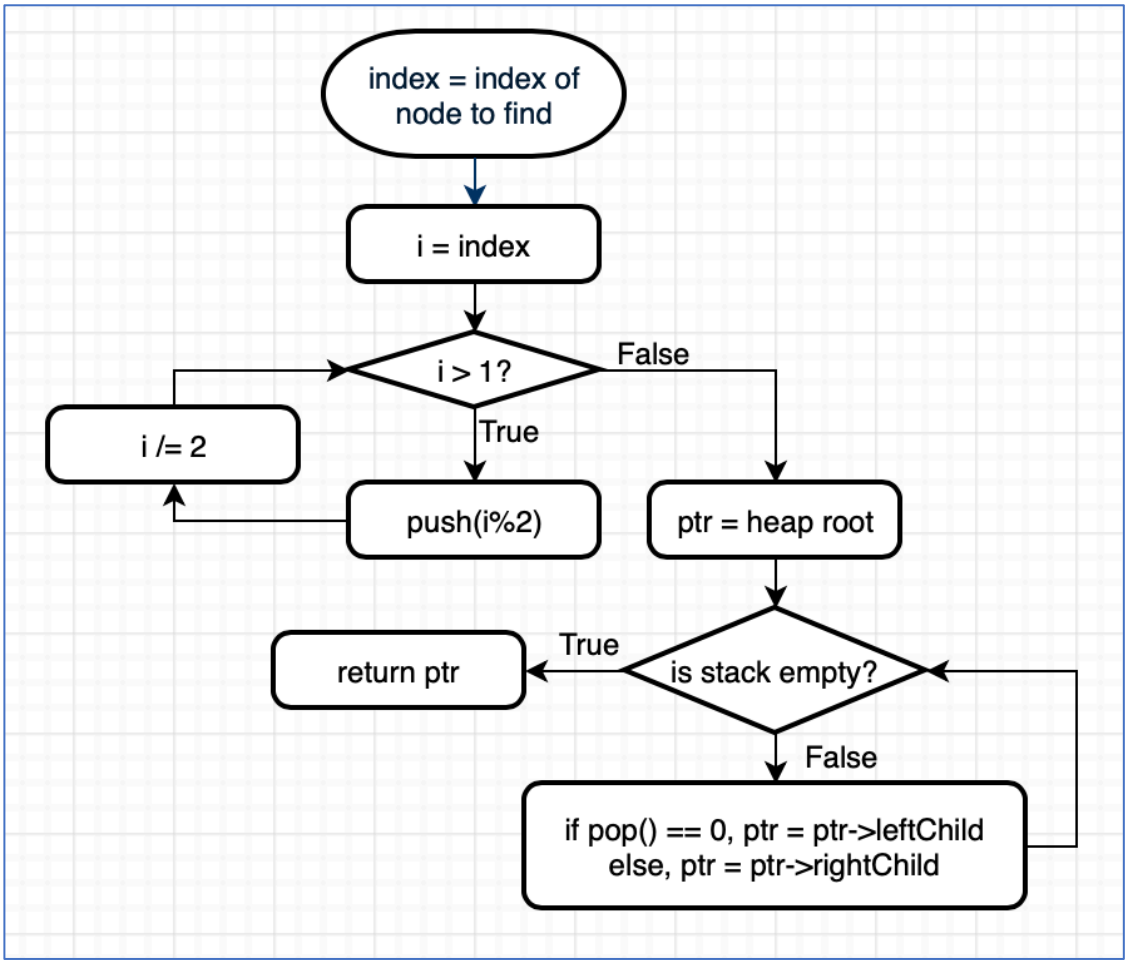
20161603
신민준

# 목차

# 1. 문제1

문제1에서는 Linked List 구조를 사용한 max heap 자료구조에서의 삽입 함수와 삭제 함수를 구현하는 것을 요구합니다. Array representation을 활용한 max heap 자료구조에서는 complete tree의 다음 element 위치를 알아내는 것은 매우 쉽지만, 이 문제에서 요구하는 Linked List 구조의 max heap에서는 다음 element의 위치를 찾아내는 것이 까다롭습니다. 이 문제를 해결하기 위해, 일반적으로 1에서부터 시작하는 complete tree의 index 값을 지속적으로 2로 나눈 나머지를 활용해, root node로부터 특정 노드까지 가는 경로를 계산했습니다.

## 1.1 프로그램 구조




**Flowchart 1. Index에 해당하는 node의 경로를 찾는 알고리즘**

## 1.2 세부 설명

이 프로그램에서 가장 중요한 문제가 되는것은 앞서 말했듯이 특정 index의 위치에 존재하는 node를 바로 찾는 것이 어렵다는 점입니다. 만약 Array representation을 사용했다면, node의 index는 Array 상에서의 index 값과 순서대로 매칭되므로 전혀 어려울 것이 없지만, Linked List 자료구조에 있어서는 index 개념이 기본적으로 존재하지 않기 때문에 구현이 까다롭습니다. 이에 대해, [Flowchart 1]에서 보인 것과 같은 알고리즘을 사용해 다음 함수 makeNext()

로 구현했습니다.

```
/**
 * Create new node in heap
 * @param  key Value for node
 * @return    Address of created node
 */
treePointer makeNext(int key) {
   treePointer ptr;
   for(int i=node_cnt+1 ; i>1 ; i/=2)
      push(i%2);
   ptr = heap;
   while(stack) {
      if(pop() == 0) {
         if(!ptr->leftChild) {
            ptr->leftChild = new_node(key, ptr);
         }
         ptr = ptr->leftChild;
      } else {
         if(!ptr->rightChild) {
            ptr->rightChild = new_node(key, ptr);
         }
         ptr = ptr->rightChild;
      }
   }
   node_cnt++;
   return ptr;
}
```

**Code 1**

이 함수는 총 노드 수를 담고 있는 전역변수인 node_cnt의 값을 사용해 [Flowchart 1]에서 소개한 알고리즘으로 다음 노드가 추가될 위치에 노드를 추가하고, 해당 노드의 주소값을 리턴합니다. 만약 heap의 root의 index 값을 1이라 한다면, 특정 index 값을 지닌 노드까지 가는 경로는 해당 index 값이 1이 될 때 까지 2로 나눈 나머지의 역순이 됩니다. 이 때, 나머지가 0이라면 해당 노드는 부모 노드의 왼쪽 노드이고, 1이라면 부모 노드의 오른쪽 노드입니다. 따라서, index 값을 2로 지속적으로 나누며 그 값을 스택에 push 했고, 이를 다시 pop 하며 heap의 root에서부터 pop된 값에 따라 자식 노드로 이동해 최종적으로 생성하고자 한 노드를 생성하고 주소값을 리턴합니다. 여기서 Stack을 조작하는 push(), pop() 함수는 일반적인 형태의 push, pop 함수와 같습니다.

이 함수에서 사용한 new_node() 함수는 새로운 노드를 생성하는 함수입니다. 이는 [Code 2]에서 확인할 수 있습니다.

```
/**
 * Create new tree node
 * @param  key    Key value for current node
 * @param  parent Parent node address. Set to NULL if this node is root node.
 * @return       Address of new tree node created.
 */
treePointer new_node(int key, treePointer parent) {
   treePointer new;

   new = malloc(sizeof(struct node));
```

```
   new->key = key;
   new->leftChild = new->rightChild = NULL;
   new->parent = parent;

   return new;
}
```

**Code 2**

구현한 makeNext() 함수를 사용해 heap 삽입 기능을 하는 insert() 함수를 생성했습니다.

```
/**
 * Insert new node into heap
 * @param  key Value for new node
 * @return     1 if key is successfully added to heap
 *             0 if key is already in heap
 */
int insert(int key) {
   treePointer ptr;

   if(search(heap, key))
      return 0;

   if(!heap) {
      heap = new_node(key, NULL);
      node_cnt++;
      return 1;
   }

   ptr = makeNext(key);

   // allign heap to meet criteria for max heap
   while(ptr->parent && ptr->parent->key < ptr->key) {
      int temp = ptr->parent->key;
      ptr->parent->key = ptr->key;
      ptr->key = temp;
      ptr = ptr->parent;
   }
   return 1;
}
```

**Code 3**

```
/**
 * Searches for key value in tree starting from root
 * @param  root Start point of the tree
 * @param  key  Value to find
 * @return      1, if key is found
 *              0, if key is not found
 */
int search(treePointer root, int key) {
   if(root) {
      if(root->key == key)
         return 1;
      else
         return search(root->leftChild, key) || search(root->rightChild, key);
   }
   return 0;
}
```

**Code 4**

[Code 4]는 [Code 3]의 insert() 함수에서 사용되는 search() 함수를 구현한 것입니다.
Insert() 함수가 호출되면, 먼저 search() 함수로 입력받은 key 값이 이미 heap에 존재하는지 확인합니다. 만일 존재한다면, 0를 리턴해 해당 값이 이미 존재함을 보입니다.
만일 heap이 비어있다면 새로운 노드를 만들고 이를 heap의 root 노드로 설정하고 1을 리턴해 성공적으로 삽입이 수행되었음을 알립니다.
위 두 경우에 모두 해당하지 않을 경우, [Code 1]에서 구현한 makeNext() 함수로 새로운 노드를 추가하고, 생성한 노드로부터 시작해 root 노드까지 max heap의 정의에 맞게 값을 오름차순으로 재정렬시켜 삽입 과정을 마무리합니다.

Insert 함수는 각 tree의 level마다의 작업을 반복하므로, 시간 복잡도는 다음과 같습니다.

$$O(\log n) \ \ when\ n = number\ of\ nodes$$

Heap의 node 삭제를 구현한 delete() 함수의 기본 개념은 통상적인 방법과 다르지 않게 구현했습니다. 즉, 먼저 root node에 index 상으로 마지막 노드를 옮기고, 새로운 root로부터 자식 node로 내려가면서 값을 재정렬하는 방식입니다. 이에 대한 구현은 [Code 5]에 보이고 있습니다.

```
/**
 * Delete a node from heap, and fetches its value.
 * @param  deleted Pointer for storing deleted key value
 * @return        0 if heap is empty, 1 if deletion was successful
 */
int delete(int* deleted) {
  treePointer ptr, last;

  if(!heap) return 0;

  last = getLast();
  *deleted = heap->key;
  if(last == heap) {
    heap = NULL;
    free(last);
    node_cnt--;
    return 1;
  }
  heap->key = last->key;
  if(last->parent->leftChild == last)
    last->parent->leftChild = NULL;
  else
    last->parent->rightChild = NULL;
  free(last);
  last = NULL;

  ptr = heap;
  while(ptr->leftChild || ptr->rightChild) {
    if(ptr->leftChild && ptr->rightChild) {
      ptr = (ptr->leftChild->key > ptr->rightChild->key) ?
      ptr->leftChild : ptr->rightChild;
    } else if(ptr->leftChild) {
      ptr = ptr->leftChild;
```

```
    } else if(ptr->rightChild) {
      ptr = ptr->rightChild;
    }

    if(ptr->parent->key < ptr->key) {
      int temp = ptr->parent->key;
      ptr->parent->key = ptr->key;
      ptr->key = temp;
    } else break;
  }
  node_cnt--;
  return 1;
}
```

**Code 5**

이 함수의 구현에 사용된 getLast() 함수는 다음과 같습니다.

```
/**
 * Get last node from heap
 * @return  last node from heap
 */
treePointer getLast(void) {
  treePointer ptr;

  for(int i=node_cnt ; i>1 ; i/=2)
    push(i%2);

  ptr = heap;
  while(stack) {
    if(pop() == 0) {
      ptr = ptr->leftChild;
    } else {
      ptr = ptr->rightChild;
    }
  }
  return ptr;
}
```

**Code 6**

먼저, delete 함수에서는 heap이 현재 비어있는지 여부를 확인하고, 비어있다면 0을 리턴합니다. 이후, getLast 함수를 이용해 현재 heap에서의 마지막 노드를 받고, 이 node로 root의 값을 대체하고, 기존 위치의 node는 삭제합니다. 이 때, 현재 heap에 node가 root node로 유일할 때와 그렇지 않을 때에 따라 약간의 방식에 차이를 두었습니다. 이렇게 heap에서 삭제가 이뤄진 후, root node에서부터 자식 노드 방향으로 내려가면서 값에 따른 재정렬을 수행합니다. 이 모든 프로시져가 이루어진 후, deleted 주소값에는 삭제된 값이 저장되며, 1을 리턴해 해당 과정이 성공적으로 이루어졌음을 보입니다.

Delete 함수 또한 insert 함수와 동일하게, tree의 각 level 마다 정해진 프로세스를 반복하므로, 시간복잡도는 다음과 같습니다.

$$O(\log n) \ \ when\ n = number\ of\ nodes$$

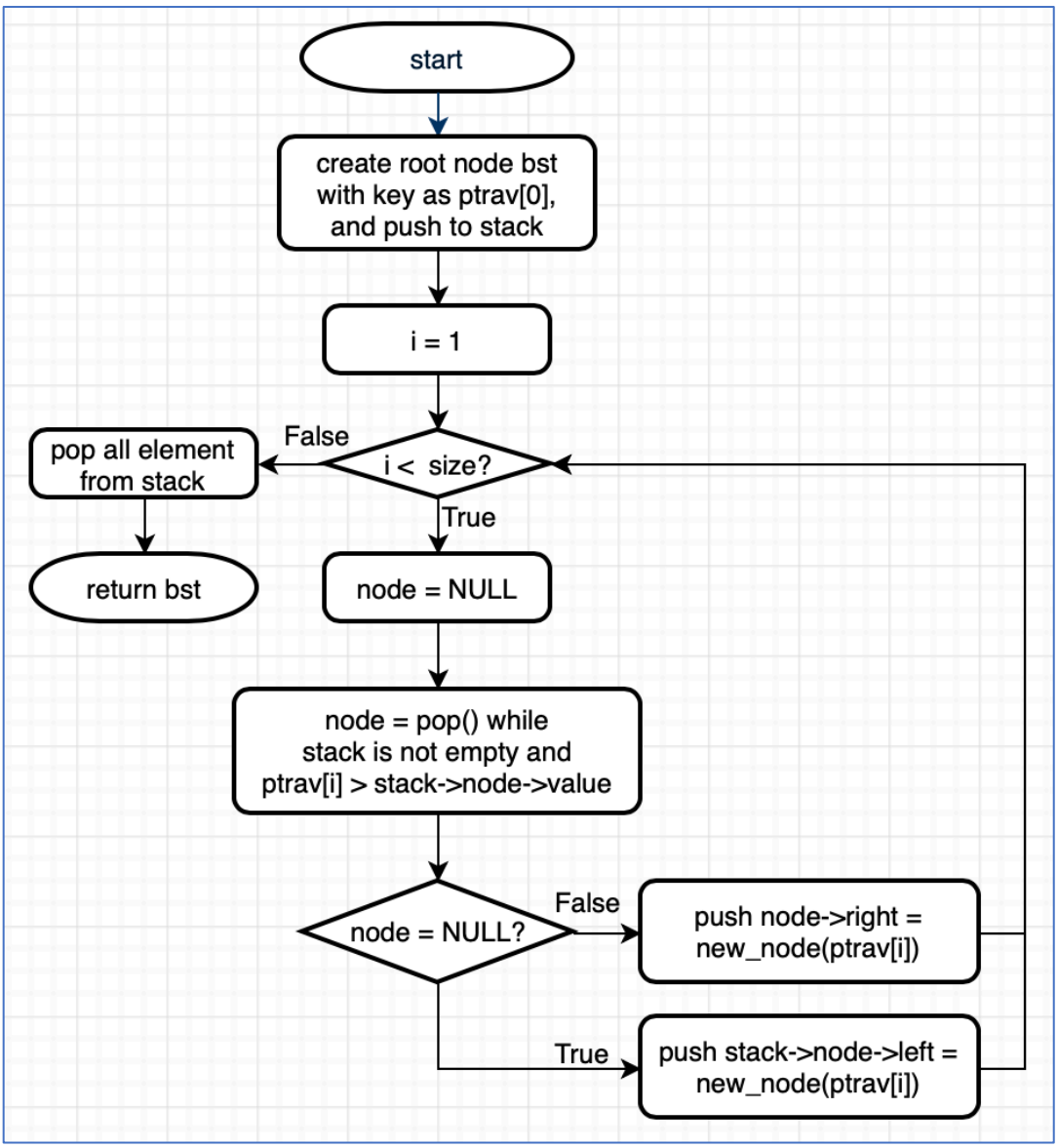
### 1.3 참고 사항

- 자료구조의 개념도 중요하지만, 실질적으로 구현할 때 고르는 자료구조에 따라 원하던 자료구조를 구현하는 난이도가 크게 차이난다는 것을 느낄 수 있었습니다. 만일 Linked List 자료구조를 사용하지 않고, Array representation 을 사용해 구현했다면, 수행시간과 메모리 사용량을 둘 다 줄이면서도 복잡하지 않고 깔끔하게 구현할 수 있었을 것입니다. 따라서, 자신의 자료구조를 구현하기에 앞서 해당 자료구조를 효율적으로 나타낼 수 있는 기본 자료구조를 어떤 것으로 사용할지에 대한 깊은 생각이 필요합니다.

## 2. 문제2

문제2는 binary search tree를 입력받은 preorder traversal로부터 생성하여, 생성한 BST에 대해 inorder/postorder traversal을 한 결과값을 출력하는 것을 요구합니다. Preorder traversal로부터 tree를 생성하는 위해선, 새로운 노드를 tree에 입력할 때 마다 해당 노드를 스택에 push 시키고, 노드를 새로 추가할 때 현재 스택의 최상위에 있는 값이 추가할 값보다 클 때 까지 pop을 시켜 해당 위치에서부터 다시 노드를 추가해가면 됩니다.

### 2.1 프로그램 구조




**Flowchart 2**

### 2.2 세부 설명

이 프로그램을 구현하는데 쓰인 자료구조는 tree와 stack입니다. 따라서, tree와 stack을 다음 [Code 7]에서 보이는 것과 같이 구조체를 Node로써 사용한 Linked List의 형태로 정의하였습니다.

```
typedef struct _tree_node {
   int value;
   struct _tree_node *left, *right;
} tree_node;

typedef struct _stack_node {
   tree_node* node;
   struct _stack_node* link;
} stack_node;

stack_node* stack = NULL;
tree_node* bst = NULL;
```

**Code 7**

Stack을 조작하는 push(), pop() 함수는 일반적인 push, pop 함수와 같습니다.

사용자의 input을 통해 ptrav[] 배열과 size 값을 받은 이후, 먼저 root node를 생성하고 이를 스택에 push합니다

```
/**
 * construct BST from preorder traversal
 * @param  ptrav Preorder traversal
 * @param  size  Length of ptrav[] array
 * @return       Root node of constructed BST
 */
tree_node* construct_BST(int ptrav[], int size) {
   tree_node* node;

   push(bst = new_node(ptrav[0]));
```

**Code 8**

Root node가 생성된 이후, ptrav[] 배열의 각 원소 값을 읽으면서 올바른 위치에 해당 값을 key로 하는 node를 삽입합니다. 올바른 위치를 찾는 방법은 [Flowchart 2]에서 보인 방법과 같고, 이를 [Code 9]처럼 구현했습니다.

```
   for(int i=1 ; i<size ; i++) {
      node = NULL;
      while(stack && ptrav[i] > stack->node->value)
         node = pop();

      if(node) {
         push(node->right = new_node(ptrav[i]));
      } else {
         push(stack->node->left = new_node(ptrav[i]));
      }
   }

   while(!stack)
      pop();

   return bst;
}
```

**Code 9**

만약 새로 추가해야 하는 ptrav[i] 값이 스택에 최상단에 있는, 즉 바로 전 loop에서 입력된 node의 key값보다 작다면, 새로 추가하는 node는 반드시 바로 전 loop에서 입력된 node의 left child로 들어가야 합니다. 따라서 스택 최상단의 노드를 읽어 그 노드의 left child로 새로운 node를 만들고 이 새로운 노드도 마찬가지로 push합니다.
반면 새로 추가하는 ptrav[i] 값이 스택의 최상단 값보다 크다면, tree의 현재 level보다 위로 올라가야 하므로 스택에서 부모 node를 pop해가며 ptrav[i] 값보다 큰 node가 스택의 최상단에 오거나, 스택이 빌 때까지 pop을 진행합니다. 이후, pop된 마지막 node의 right child로 입력받은 ptrav[i] 값의 노드를 추가하고 이 또한 push합니다.
위 과정을 모든 ptrav[] 배열의 원소에 대해 적용시키면, root node에서 시작하는 Binary Search Tree를 얻을 수 있고, 마지막으로 해당 tree의 root node를 리턴합니다.

Binary Search Tree를 생성하는 construct_BST() 함수는, 각 노드에 대해 한번의 push와 한번의 pop을 하므로, 다음과 같은 시간복잡도를 가집니다.

$$O(n), when\ n = number\ of\ nodes$$

[Code 9]에서 사용된 new_node() 함수는 다음과 같이, 간단하게 새로운 노드를 생성해 리턴해주는 역할을 합니다.

```
/**
 * Create new tree node
 * @param  value Value of the new node
 * @return       Address of the created node.
 */
tree_node* new_node(int value) {
   tree_node* new;
   new = malloc(sizeof(tree_node));
   new->value = value;
   new->left = new->right = NULL;
   return new;
}
```

**Code 10**

Binary search tree를 완성하고 나면, print_inorder(), print_postorder() 함수로 완성한 tree를 출력하도록 구현했습니다. 해당 두 함수는 [Code 11]에서 보이다시피 일반적인 형태와 다르지 않은 재귀적인 방법으로 구현했습니다.

```
/**
 * Print BST in inorder format
 * @param root BST to print
 */
void print_inorder(tree_node* root) {
   if(root) {
      print_inorder(root->left);
      printf("%d ", root->value);
      print_inorder(root->right);
   }
}
```

```
/**
 * Print BST in postorder method
 * @param root BST to print
 */
void print_postorder(tree_node* root) {
  if(root) {
    print_postorder(root->left);
    print_postorder(root->right);
    printf("%d ", root->value);
  }
}
```

**Code 11**

또한, 문제에서 추가적으로 요구했던 "BST 구성 불가" 케이스에 대해, [Code 12]에서처럼 main 함수에서 사용자로부터 입력을 받은 직후 해당 입력을 바로 검토하는 방식으로 구현했습니다.

```
  // Checking if there are duplicate value in given traversal
  for(int i=0 ; i<size ; i++)
    for(int j=i+1 ; j<size ; j++)
      if(ptrav[i] == ptrav[j]) {
        printf("cannot construct BST\n");
        return 0;
      }
```

**Code 12**

2.3 참고 사항

- BST 생성을 재귀적으로 하는 방법도 물론 존재하지만, 재귀적으로 함수를 선언하게되면 리턴 값에서 재귀가 이루어지도록 해야 Tail call optimization 최적화 기능을 사용할 수 있는데, 이런 방식으로 설계하는 일이 까다롭고, iterative하게 구현하는 것이 이해하기도 더 편한 경우였기 때문에 iterative한 방법으로 구현했습니다.
- 반대로, print_postorder와 print_inorder 함수의 경우는 재귀적으로 선언하는 것이 간단하고, 더 이해하기 편리했기 때문에 재귀적 방법으로 구현했습니다.

# 3. 문제 코드

## 3.1 문제1 코드

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct node* treePointer;
struct node {
   int key;
   treePointer parent;
   treePointer leftChild;
   treePointer rightChild;
};

typedef struct _stack_element* stack_element;
struct _stack_element {
   int value;
   struct _stack_element* link;
};

stack_element stack = NULL;
treePointer heap = NULL;
int node_cnt = 0;

/**
 * Create new tree node
 * @param  key    Key value for current node
 * @param  parent Parent node address. Set to NULL if this node is root node.
 * @return        Address of new tree node created.
 */
treePointer new_node(int key, treePointer parent) {
   treePointer new;

   new = malloc(sizeof(struct node));
   new->key = key;
   new->leftChild = new->rightChild = NULL;
   new->parent = parent;

   return new;
}
/**
 * Push value onto stack
 * @param value Integer value to push
 */
void push(int value) {
   stack_element new = malloc(sizeof(struct _stack_element));
   new->value = value;
   new->link = stack;
   stack = new;
}
/**
 * Pop value from stack
 * @return  Popped value
 */
int pop(void) {
   int value;
   stack_element popped;

   if(!stack) {
      fprintf(stderr, "Error popping empty stack\n");
      exit(1);
   }
```

```
   popped = stack;
   stack = stack->link;
   value = popped->value;
   free(popped);

   return value;
}

/**
 * Searches for key value in tree starting from root
 * @param  root Start point of the tree
 * @param  key  Value to find
 * @return      1, if key is found
 *              0, if key is not found
 */
int search(treePointer root, int key) {
   if(root) {
      if(root->key == key)
         return 1;
      else
         return search(root->leftChild, key) || search(root->rightChild, key);
   }
   return 0;
}

/**
 * Create new node in heap
 * @param  key Value for node
 * @return     Address of created node
 */
treePointer makeNext(int key) {
   treePointer ptr;
   for(int i=node_cnt+1 ; i>1 ; i/=2)
      push(i%2);
   ptr = heap;
   while(stack) {
      if(pop() == 0) {
         if(!ptr->leftChild) {
            ptr->leftChild = new_node(key, ptr);
         }
         ptr = ptr->leftChild;
      } else {
         if(!ptr->rightChild) {
            ptr->rightChild = new_node(key, ptr);
         }
         ptr = ptr->rightChild;
      }
   }
   node_cnt++;
   return ptr;
}

/**
 * Get last node from heap
 * @return  last node from heap
 */
treePointer getLast(void) {
   treePointer ptr;

   for(int i=node_cnt ; i>1 ; i/=2)
      push(i%2);

   ptr = heap;
```

```
   while(stack) {
      if(pop() == 0) {
         ptr = ptr->leftChild;
      } else {
         ptr = ptr->rightChild;
      }
   }
   return ptr;
}

/**
 * Insert new node into heap
 * @param  key Value for new node
 * @return     1 if key is successfully added to heap
 *           0 if key is already in heap
 */
int insert(int key) {
   treePointer ptr;

   if(search(heap, key))
      return 0;

   if(!heap) {
      heap = new_node(key, NULL);
      node_cnt++;
      return 1;
   }

   ptr = makeNext(key);

   // allign heap to meet criteria for max heap
   while(ptr->parent && ptr->parent->key < ptr->key) {
      int temp = ptr->parent->key;
      ptr->parent->key = ptr->key;
      ptr->key = temp;
      ptr = ptr->parent;
   }
   return 1;
}

/**
 * Delete a node from heap, and fetches its value.
 * @param  deleted Pointer for storing deleted key value
 * @return         0 if heap is empty, 1 if deletion was successful
 */
int delete(int* deleted) {
   treePointer ptr, last;

   if(!heap) return 0;

   last = getLast();
   *deleted = heap->key;
   if(last == heap) {
      heap = NULL;
      free(last);
      node_cnt--;
      return 1;
   }
   heap->key = last->key;
   if(last->parent->leftChild == last)
      last->parent->leftChild = NULL;
   else
      last->parent->rightChild = NULL;
```

```
   free(last);
   last = NULL;

   ptr = heap;
   while(ptr->leftChild || ptr->rightChild) {
      if(ptr->leftChild && ptr->rightChild) {
         ptr = (ptr->leftChild->key > ptr->rightChild->key) ?
         ptr->leftChild : ptr->rightChild;
      } else if(ptr->leftChild) {
         ptr = ptr->leftChild;
      } else if(ptr->rightChild) {
         ptr = ptr->rightChild;
      }

      if(ptr->parent->key < ptr->key) {
         int temp = ptr->parent->key;
         ptr->parent->key = ptr->key;
         ptr->key = temp;
      } else break;
   }
   node_cnt--;
   return 1;
}

int main(int argc, const char* argv[]) {
   char cmd;
   int key, result;
   while(1) {
      scanf("%c", &cmd);
      if(cmd == 'q') {
         // freeing all nodes
         while(heap)
            delete(&result);
         break;
      }

      if(cmd == 'i') {
         scanf("%d", &key);
         if(insert(key)) {
            printf("Insert %d\n", key);
         } else {
            printf("Exist number\n");
         }
      } else if(cmd == 'd') {
         if(delete(&result)) {
            printf("Delete %d\n", result);
         } else {
            printf("The heap is empty\n");
         }
      }
   }
   return 0;
}
```

### 3.2 문제2 코드

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct _tree_node {
   int value;
   struct _tree_node *left, *right;
} tree_node;

typedef struct _stack_node {
   tree_node* node;
   struct _stack_node* link;
} stack_node;

stack_node* stack = NULL;
tree_node* bst = NULL;

/**
 * Push tree node into stack
 * @param node Node to push
 */
void push(tree_node* node) {
   stack_node* new = malloc(sizeof(stack_node));
   new->node = node;
   new->link = stack;
   stack = new;
}

/**
 * Pop tree node from stack
 * @return  Popped tree node
 */
tree_node* pop(void) {
   stack_node* popped;
   tree_node* node;

   if(!stack) {
      fprintf(stderr, "Popping empty stack.\n");
      exit(1);
   }

   popped = stack;
   node = popped->node;
   stack = stack->link;
   free(popped);
   return node;
}

/**
 * Create new tree node
 * @param  value Value of the new node
 * @return       Address of the created node.
 */
tree_node* new_node(int value) {
   tree_node* new;
   new = malloc(sizeof(tree_node));
   new->value = value;
   new->left = new->right = NULL;
   return new;
}

/**
```

```
 * construct BST from preorder traversal
 * @param  ptrav Preorder traversal
 * @param  size  Length of ptrav[] array
 * @return       Root node of constructed BST
 */
tree_node* construct_BST(int ptrav[], int size) {
   tree_node* node;

   push(bst = new_node(ptrav[0]));

   for(int i=1 ; i<size ; i++) {
      node = NULL;
      while(stack && ptrav[i] > stack->node->value)
         node = pop();

      if(node) {
         push(node->right = new_node(ptrav[i]));
      } else {
         push(stack->node->left = new_node(ptrav[i]));
      }
   }

   while(!stack)
      pop();

   return bst;
}

/**
 * Print BST in inorder format
 * @param root BST to print
 */
void print_inorder(tree_node* root) {
   if(root) {
      print_inorder(root->left);
      printf("%d ", root->value);
      print_inorder(root->right);
   }
}

/**
 * Print BST in postorder method
 * @param root BST to print
 */
void print_postorder(tree_node* root) {
   if(root) {
      print_postorder(root->left);
      print_postorder(root->right);
      printf("%d ", root->value);
   }
}

/**
 * Free BST from memory
 * @param root BST to free
 */
void free_BST(tree_node* root) {
   if(root) {
      free_BST(root->left);
      free_BST(root->right);
      free(root);
   }
}
```

```c
int main(int argc, const char* argv[]) {
   int size, *ptrav;

   scanf("%d", &size);

   ptrav = malloc(sizeof(int)*size);
   for(int i=0 ; i<size ; i++) {
      scanf("%d", &ptrav[i]);
   }


   // Checking if there are duplicate value in given traversal
   for(int i=0 ; i<size ; i++)
      for(int j=i+1 ; j<size ; j++)
         if(ptrav[i] == ptrav[j]) {
            printf("cannot construct BST\n");
            return 0;
         }

   construct_BST(ptrav, size);

   printf("Inorder: ");
   print_inorder(bst);
   printf("\n");
   printf("Postorder: ");
   print_postorder(bst);
   printf("\n");

   free_BST(bst);

   return 0;
}
```