본 PSet 은 저의 강의 경험과 학생들의 의견 및 Stanford CS106 과 Harvard CS50 같은 강의에서 수집된 자료를 토대로 작성되었습니다. 본 PSet 에 문제가 있거나, 질문 혹은 의견이 있다면, 언제든지 알려 주시면 감사하겠습니다. 강의 개선에 많은 도움이 되겠습니다.
idebtor@gmail.com

PSets Heapsort & PSet Heap & PQ

목차

힙

힙은 다음과 같은 힙 속성을 충족하는 트리 기반 데이터 구조입니다. A 가 B 의 부모 노드인 경우 노드 A 의 키가 노드 B 의 키에 따라 힙 전체에 동일한 순서로 정렬됩니다.

힙의 일반적인 구현은 이진 힙으로, 트리는 배열 구조의 완전 이진 트리입니다(complete binary tree).

최대 힙에서 부모 노드의 키는 항상 자식 노드의 키보다 크거나 같고, 가장 높은 키가 루트 노드에 위치합니다. 최소힙에서 부모 노드의 키는 항상 자식 노드의 키보다 작거나 같고, 가장 작은 키가 루트 노드에 위치합니다.

최대힙 vs. 최소힙

알다시피 최소힙은 sink 와 swim 에 사용되는 비교 방식을 제외하고 최대힙과 알고리듬과 데이터 구조가 동일합니다. 최대힙에서 'less()'함수를 꾸준히 사용해 왔습니다. 최대힙과 최소힙 코드의 유일한 차이점은 less() 혹은 more() 사용 여부입니다. 놀랍지 않나요?

함수마다 함수를 복제하는 대신, 힙 구조의 현재 작업에 따라 less() 혹은 more() 함수를 가리키는 함수 포인터를 유지합니다. 다시 말해, 모든 코드에서 less() 또는 more()을 사용하는 대신, less() 혹은 more()을 가리키는 함수 포인터를 사용합니다.

이를 위해 'comp' 함수 포인터를 heap.h 에 정의할 수 있습니다.

```
struct Heap {                          // heap or min/max priority queue
  int *nodes;                          // an array of nodes
  int capacity;                        // array size of node or key, item
  int N;                               // number of nodes in the heap
  bool(*comp)(heap, int, int);         // less() for max, more() for minheap
  Heap(int capa = 2) {
     capacity = capa;
     nodes = new int[capacity];
     N = 0;
     comp = nullptr;
  };
  ~Heap() {};
};
using heap = Heap *;
```

사용자가 최대힙을 선택하면 이 함수 포인터는 less()로 설정되고, 그렇지 않을 경우 more()로 설정됩니다. less()를 사용할 때마다 'comp' 함수 포인터로 대체합니다. 즉, 하드 코딩된 less() 함수를 사용하는 대신 'comp' 함수 포인터를 사용합니다. 그러면 CBT 는 사용자의 선택에 따라 최대힙 또는 최소힙이 됩니다.

사용자가 'z' 옵션을 선택하면 setType() 함수를 호출하여 비교 함수 포인터(p->comp = ?)를 설정할 수 있습니다.

```
// sets the compare function less() for maxheap, more() for minheap.
void setType(heap p, bool maxType) {
  p->comp = maxType ? ::less : more;    // less() uses global scope resolution ::
}
```

우선순위 큐

우선순위 큐는 일반 큐 혹은 스택 데이터 구조와 유사하지만 각 요소가 "우선순위"를 갖는 추상 자료형입니다. 우선순위 큐에서는 우선순위가 높은 요소가 우선순위가 낮은 요소 앞에 위치합니다.

힙은 우선순위 큐라고 하는 추상 자료형을 최대한 효율적으로 구현한 것이며, 실제로 구현 방법과 관계없이 **우선순위 큐**를 종종 "힙"이라고 합니다.

힙 정렬

힙 정렬은 비교 기반 정렬 알고리듬입니다. 힙 정렬은 개선된 선택 정렬(selection sort)이라고 볼 수 있습니다. 선택 정렬의 알고리듬처럼 입력값을 정렬된 영역과 정렬되지 않은 영역으로 나누고, 가장 큰 요소를 정렬된 영역으로 옮기며 정렬되지 않은 영역을 반복적으로 축소합니다. 선형 시간 탐색 대신 **힙 데이터 구조를** 사용하여 최댓값을 찾는 방식으로 개선되었습니다.

힙 정렬 알고리듬

힙 정렬 알고리듬은 먼저 리스트를 **최대힙**으로 만들어 준비합니다. 그런 다음 알고리듬은 리스트의 첫 번째 값과 마지막 값을 반복적으로 교환하여 힙 작업에 고려되는 값의 범위(N)을 1 만큼 줄이고, 새 첫 번째 값(루트)을 힙에 위치한 자리로 이동합니다. 이 작업은 고려되는 값의 범위가 길이가 한 개의 값이 될 때까지 반복합니다.

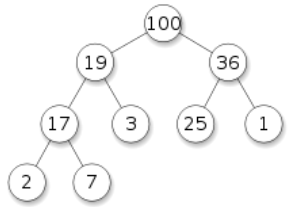
단계는 다음과 같습니다:

1. 입력 데이터로 최대힙/최소힙을 빌드합니다 (이 작업을 **heapify** 라고 합니다). 이 과정은 리스트에서 힙을 O(n) 작업으로 빌드합니다.
2. 리스트의 첫 번째 요소와 마지막 요소를 교환합니다. 리스트의 범위를 1 씩 줄입니다.
3. 리스트에서 sink() 함수를 호출하여 새 첫 번째 요소를 힙의 적절한 인덱스로 분리합니다.
4. 리스트의 범위가 요소 하나가 아닌 경우 step (2)로 갑니다.

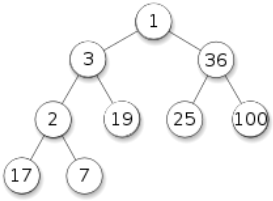
heapify() 작업은 한 번 실행되며 O(n) 작업입니다. sink() 함수는 O(log(n))이며 n 번 호출됩니다. 따라서 이 알고리듬의 성능은 O(n + n * log(n))로, O(n log n)과 같습니다.

Heapify

Heapify 는 완전 이진 트리를 힙 데이터 구조로 변환하는 작업입니다. 힙은 완전 이진 트리여야 하며 힙 순서 속성을 충족해야 합니다. 각 노드에 저장된 값이 자식 노드의 값보다 크거나 같아야 합니다.



다음은 최대힙 데이터 구조입니다 (루트 노드가 가장 큰 값을 포함합니다). 이 힙을 포함하는 배열은 {100, 19, 36, 17, 3, 25, 1, 2, 7}로 표시합니다. 완전 이진 트리는 각 부모 노드가 자식 노드보다 크거나 같은 힙 순서를 유지하며 왼쪽 정렬된다는 점을 기억하세요.



위 힙 구조에 도달하기 위해 {1, 3, 36, 2, 19, 25, 100, 17, 7}와 같은 이진 트리에서 시작할 수 있습니다.

heapify()에서 sink()는 **마지막 부모 노드(이 예시에서는 마지막 내부 노드 또는 2)에서 거슬러 올라가며 작업하여** 부모 노드를 각 자식 노드와 반복하여 비교하며, 자식 노드가 더 크면 최대힙 데이터 구조가 완성될 때까지 부모 노드와 자식 노드를 교환합니다.

Step 1.2: 힙 구축 및 힙 정렬 추적

힙 데이터 구조 및 힙 정렬에 대해 알아봅시다. 먼저 알고리듬과 작동 방식을 이해하는 것이 중요합니다. 힙 정렬 알고리듬은 두 단계로 구성되어 있습니다. 자세한 내용은 위에 설명한 힙 정렬 알고리듬을 참고하세요.

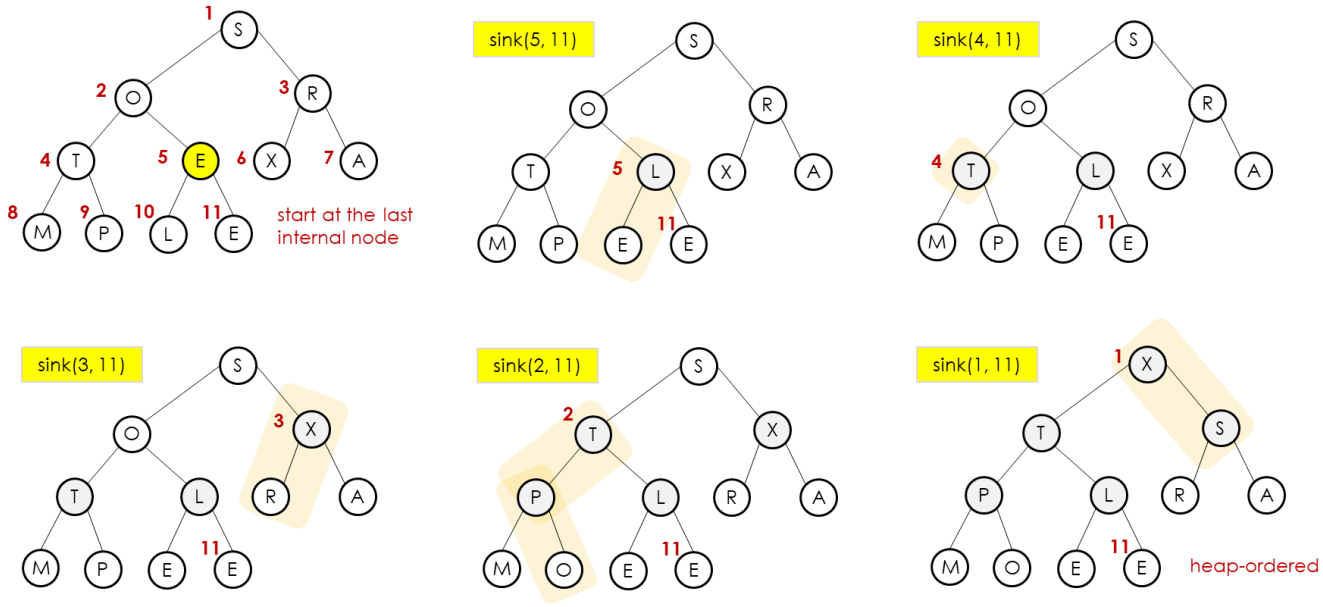
아래와 같이 작은 입력 리스트인 list a[]가 있다고 가정합시다. 간소화를 위해 첫 번째 요소인 a[0]을 제외합니다.

```
a[] = {' ', 'S', 'O', 'R', 'T', 'E', 'X', 'A', 'M', 'P', 'L', 'E'};
```

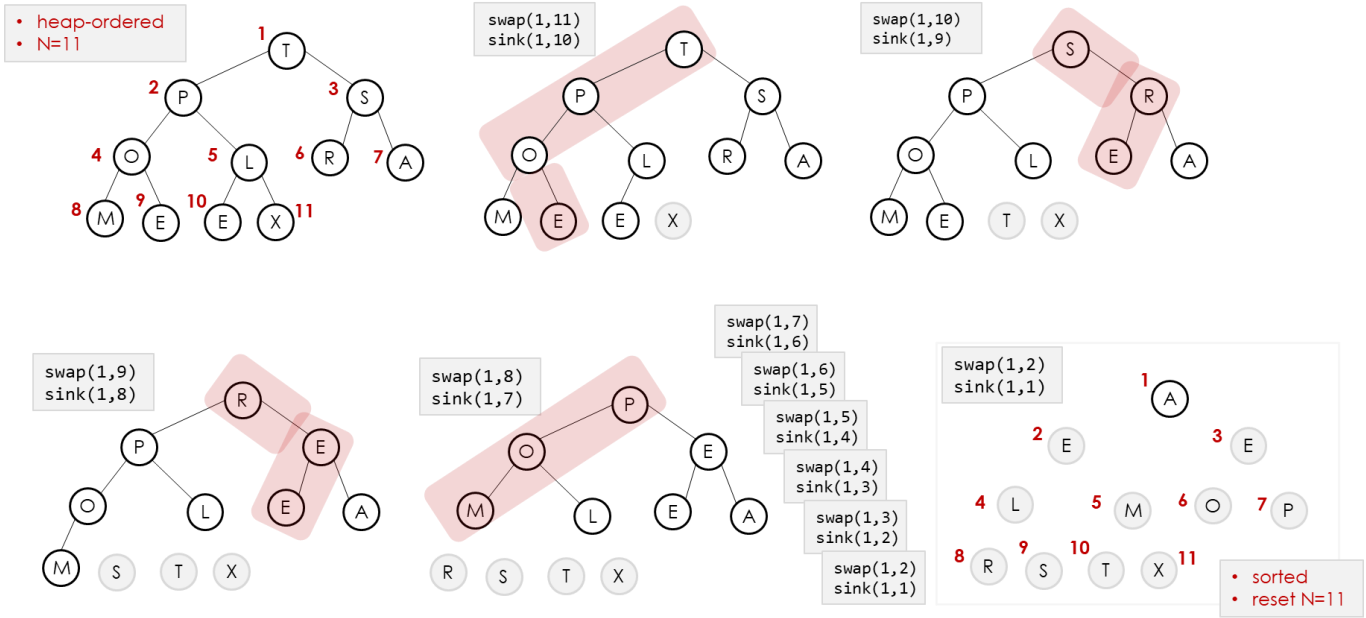
첫 번째 단계는 입력 배열 a[]을 최대 힙으로 만듭니다. 이 과정을 sink()를 반복적으로 사용하는 '**heapify**'라고 부릅니다. 두 번째 단계는 정렬된 결과를 제자리에 두는 것입니다.

정렬하는 동안 바뀌는 트리 구조를 모두 그려봅시다. 각 트리에서 N 과 k 의 값과 정렬된 결과 상태를 기록합니다. 다시 말해, 다음 그림과 비슷하게 기록하여 제출하세요. 제공된 문자열의 heapify 과정을 손으로 그려 이미지 파일로 제출하세요.

Heapsort – 1st Pass: Build heap using bottom-up method (heapify)



Heapsort – 2nd Pass: Repeatedly remove the maximum key.



1. 첫 번째로 할 일은 위에 보이는 것처럼 손으로 힙 정렬을 수행하는 것입니다.
   - **"CHRISTIAN" 뒤에 본인 이름의 이니셜 두 글자**를 붙인 단어를 사용하세요.
     예를 들어, 제 이름의 이니셜은 YS 이므로 제가 작업해야 할 단어는 "CHRISTIANYS"입니다.
     자신의 이름이 "한, 동만"이라면 단어는 "CHRISTIANDM"이 됩니다.
2. 질문:

A. 첫 번째 패스에서는 작업할 첫 번째 문자와 두 번째 문자(본인의 이니셜), 그리고 각각 heapify 하는 동안 몇 번 비교하는지 작성하세요:
   1st pass _____, _____ comparisons _____, _____ comparisons
B. 두 번째 패스에서는 sink 할 첫 번째 문자와 두 번째 문자, 그리고 각각 sink 하는 동안 몇 번 비교하는지 작성하세요:
   2nd pass: _____, _____ comparisons _____, _____ comparisons

예를 들어, "Y"와 "S"의 경우 다음과 같습니다.

A. 1st pass: "Y" _____ comparisons, "S" _____ comparisons
B. 2nd pass: "Y" _____ comparisons, "S" _____ comparisons

- 위의 두 패스의 패턴을 따라가면서 _ _에 본인의 이니셜을 넣은 단어("CHRISTIAN_ _")의 힙 정렬 과정을 그리세요.
- 위의 질문과 답변을 함께 제출하세요.
- 이 step 의 답변이 step 1.2 혹은 heapsortx.exe 의 결과와 일치하는지 확인하세요. 일치하지 않는다면 이 step 에 대한 점수를 얻을 수 없습니다.

Step 1.2: heapsort.cpp 에 힙과 힙 정렬 구현

swap(), less(), more(), sink(), swim(), 그리고 heapify()와 같은 몇 가지 도우미 함수를 먼저 **구현**합니다. 그런 다음 이 도우미 함수들을 사용하여 heapsort(), grow(), 그리고 trim() 함수를 구현하여 테스트합니다. jumpstart 용으로 제공된 heapsortx.exe 와 같이 작동하도록 만드세요.

코드 작성을 마치면, 단어(CHRISTIAN_ _)로 테스트해 보세요. 뼈대 코드 파일(heapsort.cpp)을 코딩할 때, 제공된 지침이 있다면 **지침을 따라** 코딩하세요.

주어진 입력 리스트를 사용하여 테스트합니다.

- heapsort.cpp - 모든 작업(코드 작성)은 이 파일에서 수행합니다.
- heapsortx.exe - 테스트용 실행 파일, 첫 번째 문자는 공백입니다.
- 힙 정렬용 명령행 빌드:

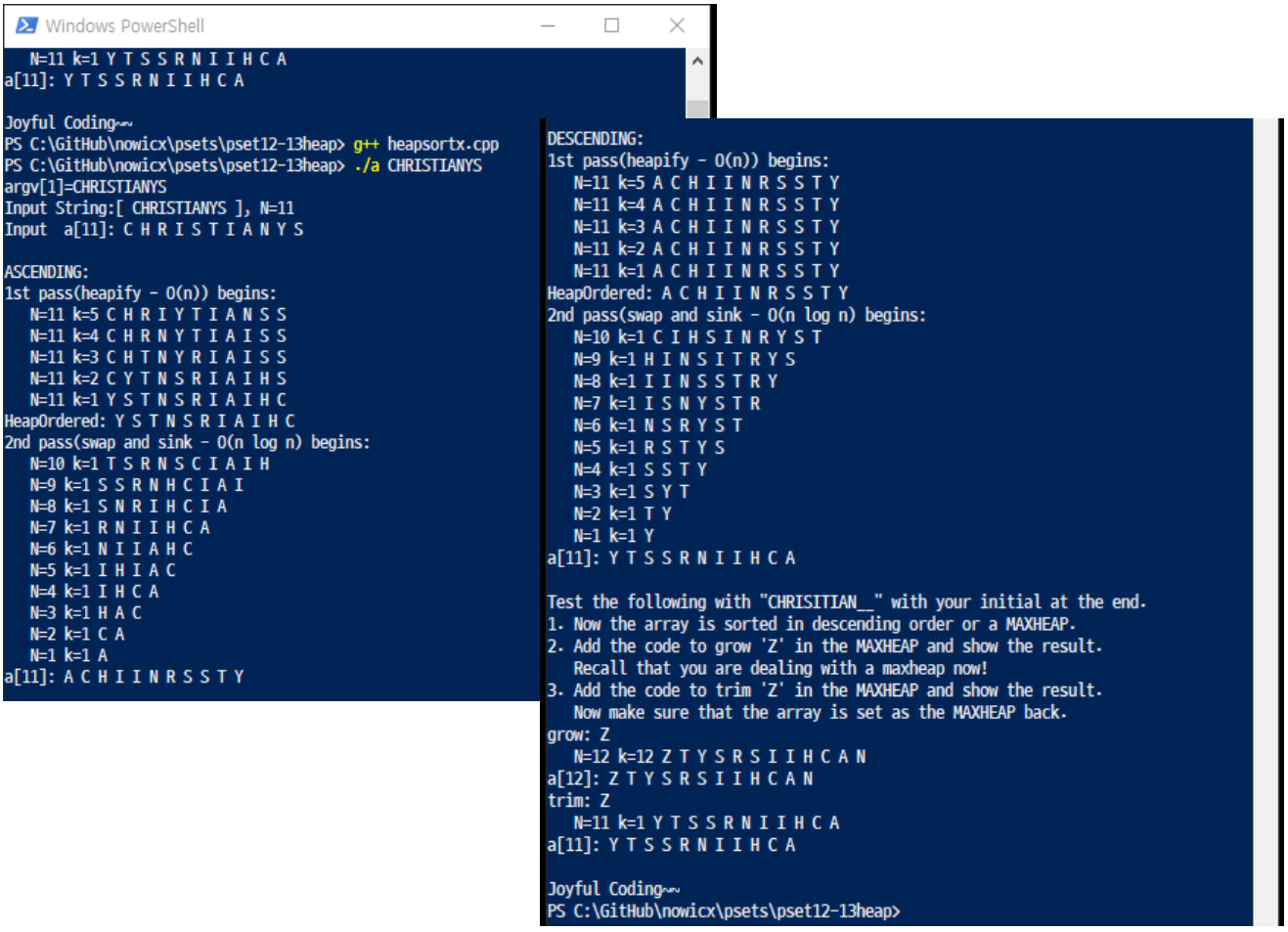
```
$ g++ heapsort.cpp –o heapsort                  #pc
$ g++ -std=c++11 heapsort.cpp –o heapsort       #mac
```

heapsort.cpp 는 다른 외부 파일이나 라이브러리와 연동하지 않습니다. 완전한 힙 & PQ 를 구현하기 전에 힙 데이터 구조와 알고리듬을 이해할 수 있는 좋은 기회입니다.

힌트: :: less()를 사용하여 오름차순에 대한 heapsort()를 먼저 구현해 보세요.

1. 오름차순으로 작동하는지 확인한 후 sink()와 swim()에서 less() 함수가 사용된 부분을 comp 함수 포인터로 대체합니다.
   간소화를 위해 이 파일에서 comp 는 전역 변수로 정의되어 있습니다.
2. main()에서 heapsort()를 호출하기 전에 오름차순의 경우 comp 를 less 로, 내림차순의 경우 comp 를 more 로 설정합니다.
3. 일반적으로, 전역 변수를 사용하지 않아야 합니다. 그러나 이 예시에서는 편의를 위해 전역 변수 comp 를 사용합니다.
4. _ _에 본인의 이니셜을 넣은 단어("CHRISTIAN_ _")로 heapsort.cpp 를 테스트합니다.
5. 본인의 단어("CHRISTIAN_ _")로 heapsort.cpp 를 테스트하고 grow("Z")와 trim("Z")를 합니다.

실행 예시:



완전 이진 트리(CBT – Complete Binary Tree) & 힙

준비 운동으로 힙이라는 프로젝트를 빌드하고 완전 이진 트리를 출력하세요. 다음 파일들이 제공됩니다. 이 step 에서 준비 운동 단계에서 사용했던 **heapsort.cpp 를 포함하지 않아야** 합니다. 평소와 같이 nowic.h 와 nowic.lib 을 사용하세요.

- heap.h, tree.h - 수정 금지
- heap.cpp - heapprint()를 제외한 모든 작업을 구현하는 파일
- heapDriver.cpp - 반드시 필요한 경우에만 수정
- heapprint.cpp - heapprint()를 구현하는 파일
- treeprint.cpp - 수정 금지
- heapx.exe - 참고용 DEBUG off 버전의 실행 파일

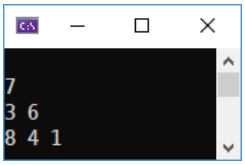
- 명령행에서 빌드하세요. heapsort.cpp 를 포함하지 **마세요**. mac 은 **–std=c++11** 를 추가하세요:

```
$ g++ heap.cpp heapDriver.cpp heapprint.cpp treeprint.cpp –I../include –L../lib –lnowic –o heap
```

Step 2.1: growCBT(), trimCBT(), contains() 및 reserve() 사용

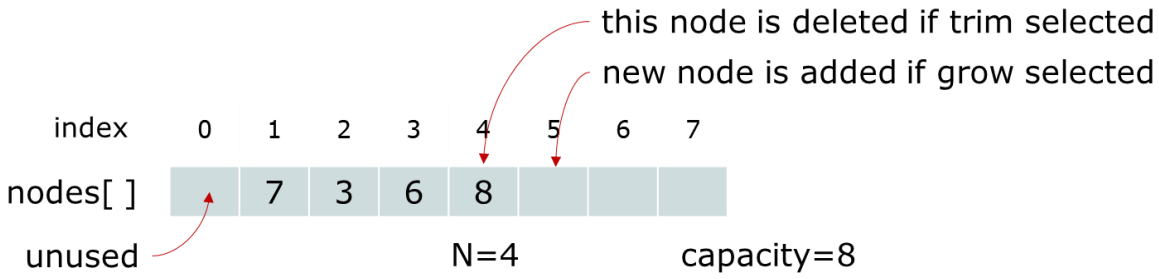
알다시피 완전 이진 트리는 경우에 따라 맨 아래 레벨을 제외하고는 트리의 각 레벨이 채워져야 합니다. 맨 아래 레벨 또한 왼쪽에서 오른쪽으로 채워져야 합니다. 'trim' 메뉴를 선택할 경우, 트리의 마지막 리프 노드를 삭제합니다.

(1) growCBT(), trimCBT(), 그리고 contains()를 먼저 구현합니다.
CBT 메뉴에서 'g'와 't' 메뉴를 선택하면 각각 growCBT()와 trimCBT() 함수를 호출합니다. 새 키를 입력하면 "nodes" 배열이라는 힙 구조 멤버에 키를 삽입해야 합니다. 트리의 마지막 위치에 삽입합니다. "trim"에서는 트리의 마지막 노드를 자동으로 삭제합니다.

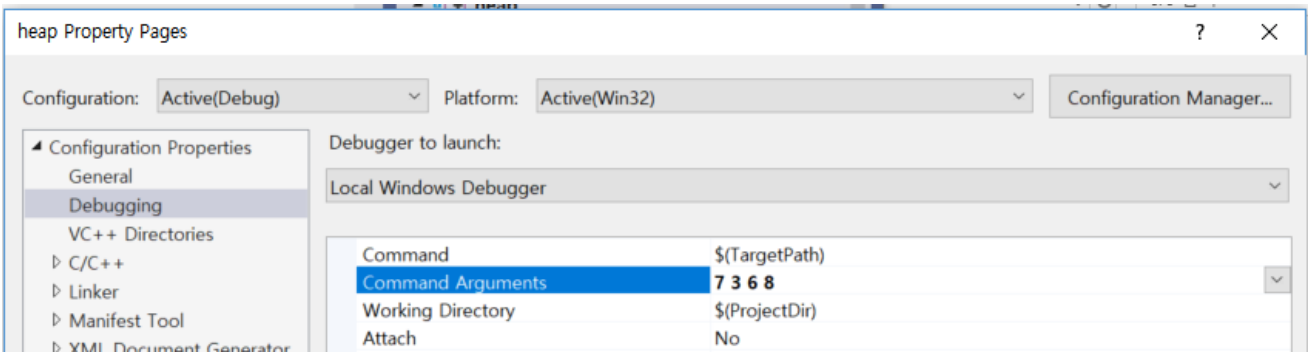


키를 입력한 후, 예를 들어 7 3 6 8 4 1을 하나씩 입력한 후, 힙의 nodes[]가 아래와 같이 메모리에 저장됩니다.

힙에 7, 3, 6, 8 이 있는 경우 힙의 설정은 다음과 같습니다.



명령행 인수를 7, 3, 6, 8 로 초기화하면 이런 상태가 됩니다. Visual Studio 에서, project properties → Configuration Properties → Debugging → Command Arguments 로 이동하세요.



(2) 많은 노드를 확장하거나 축소할 때 growCBT()와 trimCBT()가 동일하게 작동하도록 제공된 reserve() 함수를 사용합니다.

이제 트리를 grow 하거나 트리에 더 많은 노드를 삽입할 때, 프로그램 시작 시 주어진 샘플 트리와 동일한 크기로 트리를 생성했으므로 문제가 발생할 수 있습니다. 이 문제를 먼저 해결해야 합니다.

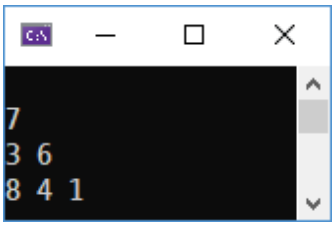
먼저, 힙과 새 용량(보통 N * 2 또는 N/2)을 받아 힙의 노드 배열을 유지하는 reserve() 함수를 구현합니다.

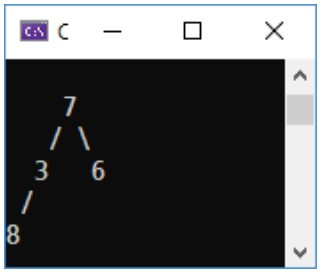
두 번째로, 필요한 경우 노드의 배열 크기를 조정하기 위해 growCBT()와 trimCBT()에서 reserve() 함수를 호출해야 합니다. 이를 수행할 적절한 때를 찾아야 합니다. 다음과 같은 원리를 적용할 수 있습니다.

노드 배열의 크기(노드의 개수, N)가 용량(또는 배열 할당 크기)에 도달하면 노드 배열의 크기를 두 배로 늘립니다. 크기(또는 N)가 용량의 ¼에 도달하면 배열 크기(용량)을 절반으로 줄입니다.

Step 2.2: heapprint.cpp 에 heapprint() 구현

heapprint_level() 함수에 단순히 레벨 별로 코드가 작성된 힙 구조를 출력하는 게 아니라 트리처럼 출력하고자 합니다.





출력을 위해 힙 데이터 구조에서 **트리를 구성해야 합니다**. 트리를 구성하면 **treeprint()** 함수를 사용하여 트리를 출력할 수 있습니다.

힙(완전 이진 트리)에서 이진 트리를 생성하는 방법이 2 가지 있습니다.

방법 1: 큐 사용

이 알고리듬은 큐를 사용하여 정수형 배열로 된 완전 이진 트리에서 이진 트리를 빌드합니다.

0. CBT 의 크기가 0 이면 nullptr 을 반환합니다.
1. CBT 의 첫 번째 키 (또는 nodes[1])로 트리 (루트) 노드를 생성합니다.
2. 트리 루트 노드를 큐에 더합니다.
3. **CBT nodes[2]부터 nodes[N]까지** 루프를 수행합니다.
   A. CBT nodes[]로부터 새 노드를 생성합니다.
   B. 큐의 트리 노드를 가져옵니다.
   C. 이 트리 노드의 왼쪽 자식 노드가 존재하지 않으면,
      새 노드를 트리 노드의 왼쪽 자식 노드로 설정합니다.
      이 트리 노드의 오른쪽 자식 노드가 존재하지 않으면,
      새 노드를 트리 노드의 오른쪽 자식 노드로 설정합니다.
   D. 트리 노드가 이미 가득 찬 경우, pop(또는 큐에서 제거) 합니다.
   E. 새 노드를 큐에 더합니다 (자식 노드가 있는 경우 나중에 추가).
4. treeprint(root)

```
// Using queue, build binary tree from CBT
tree buildBT(heap p) {
 std::queue<tree> que;
 int N = size(p);
 tree root = new TreeNode{ p->nodes[1] };
 que.push(root);

  // your code here
```

```
}
```

위의 heapprint.cpp 에 있는 buildBT()를 구현합니다.

방법 2: 재귀 사용

이 알고리듬은 재귀를 사용하여 정수형 배열로 된 완전 이진 트리에서 이진 트리(BT)를 빌드합니다. 우리는 이전에 정수형 배열로 AVL 트리를 만든 경험이 있습니다. 여기에 유사한 알고리듬을 적용할 수 있습니다. AVL 과 BT 의 유일한 차이점은 재귀 중에 패스할 배열의 범위입니다. buildAVL() 함수를 복습해 보세요.

먼저, 정수형 배열에서 완전 이진 트리를 만드는 재귀 함수를 생성합니다. 이 함수는 정수형 배열, 시작 인덱스, 그리고 배열의 크기를 사용하여 아래와 같이 루트를 반환합니다.

```
tree buildBT(int *nodes, int i, int n) {
If i > n, return nullptr – terminate condition
Create the tree (root) node with nodes[i]).
Invoke buildBT() for all its left children (or i * 2).
Set its return to the left child of the root.
Invoke buildBT() for all its right children (or i * 2 + 1).
Set its return to the right child of the root.
Return root
}
```

위의 heapprint.cpp 에 있는 buildBT()를 구현합니다.

방법 1 & 방법 2 테스트하는 방법

buildBT() 구현을 **마치면**, 그 이후의 작업은 식은 죽 먹기입니다. 배움을 목적으로, 트리의 크기가 짝수라면 재귀로 만든 트리를 사용하고, 그렇지 않으면 반복으로 만든 트리를 사용합니다. 아래와 같이 테스트해 보세요.

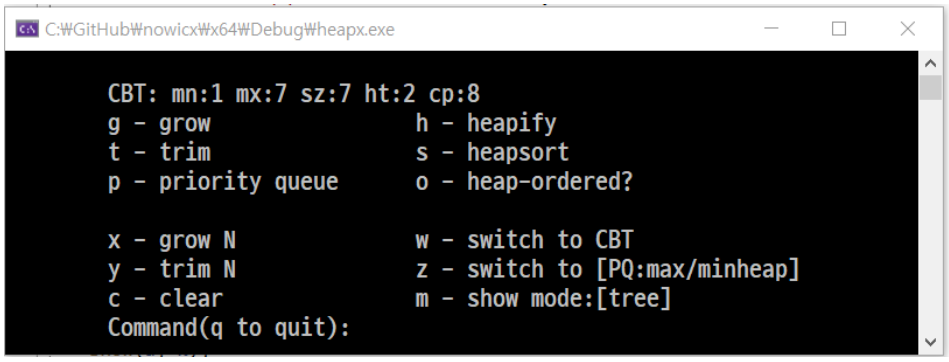
트리를 출력한 후 clear()를 호출하는 것을 잊지 마세요.

```
// print a heap using treeprint() - must build a tree to call treeprint()
void heapprint(heap p, int mode) {
 if (empty(p)) return;

  // build tree in two different ways for pedagogical purpose
 if (size(p) % 2 == 0) {
    cout << "\t[Tree built using recursion]\n";
    root = buildBT(p->nodes, 1, size(p));  // using recursion
  }
 else {
    cout << "\t[Tree built using iteration]\n";
    root = buildBT(p);                  // using iteration
 }

 switch (mode) {
```

```
 case TREE_MODE:
    treeprint(root);
    break;
 case LEVEL_MODE:
    treeprint_levelorder(root);
    break;
 default: // TASTY_MODE: show the fist and last few levels only
    treeprint_levelorder_tasty(root);
    cout << endl;
 }
  clear(root);
}
```

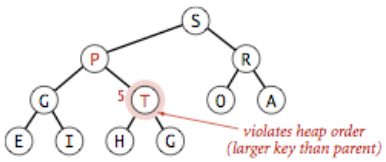
Step 2.3: 우선순위 큐(PQ): 최대힙/최소힙

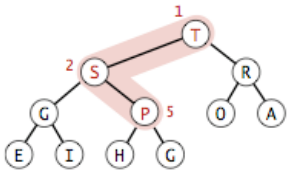
이 단계에서는 다음 메뉴에 주어진 대로 힙 추상 자료형을 구현합니다.



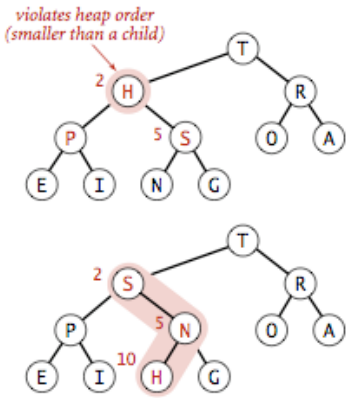
grow 또는 insert: 메뉴 옵션 **g** 를 구현합니다. 배열의 끝(또는 마지막 리프 노드)에 새로운 항목을 **추가**하고, 힙의 크기를 증가한 다음, 힙의 상태를 되찾기 위해 해당 항목을 가지고 힙을 거슬러 올라갑니다(**swim up**).

swim (Bottom-up reheapify): 노드의 키가 부모 노드의 키보다 커져서 힙 순서를 위반하는 경우, 노드와 그 부모 노드를 교환하여 문제를 해결할 수 있습니다. 교환 후 노드가 두 자식 노드보다는 크지만 (하나는 이전 부모 노드이고 다른 하나는 이전 부모의 자식 노드이므로 이전 부모 노드보다 작음), 노드가 여전히 새로운 부모 노드보다 클 수도 있습니다. 이런 문제 역시 동일한 방식으로 해결할 수 있습니다. 더 큰 키 또는 루트가 있는 노드에 도달할 때까지 힙의 위로 이동합니다.





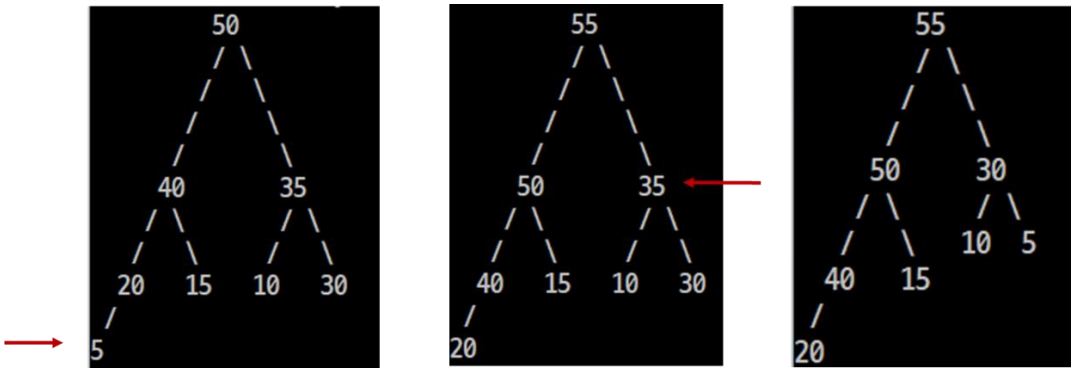
trim 또는 delete: 메뉴 옵션 **t** 를 구현합니다. 최대힙에서 최대값 또는 루트를 제거합니다. 맨 위에서 가장 큰 항목을 제거하고, 힙의 맨 끝에 위치한 항목을 맨 위에 놓고, 힙의 크기를 줄인 다음, 힙의 상태를 되찾기 위해 해당 항목을 가지고 힙을 거슬러 내려갑니다 (sink down).



sink (Top-down heapify): 노드의 키가 자식 노드 하나 또는 둘 다보다 작아져 힙 순서를 위반하는 경우, 노드를 두 자식 노드 중 더 큰 노드와 교환하여 문제를 해결할 수 있습니다. 교환 후자식 노드 사이에서 힙 순서를 위반하는 경우가 생길 수 있습니다. 이런 문제 역시 동일한 방법으로 해결합니다. 두 자식 노드 모두가 더 작은 곳에 이르거나 맨 아래에 도달할 때까지 힙의 아래로 이동합니다.

우선순위 큐 (교체): 사용자는 힙의 키값 중 하나를 변경할 수 있습니다. 새 노드는 그 값과 힙의 유형에 따라 힙의 위로 올라갈 수도, 힙의 아래로 내려갈 수도 있습니다.

예시:
5 를 55 로 바꾸면, 55 는 힙의 위로 올라가 루트에 도달하고, 20 은 맨 아래에 배치합니다.
35 를 5 로 바꾸면, 30 은 35 가 있는 곳까지 올라가고, 5 가 오른쪽 코너로 내려갑니다.



Step 2.4: growN() & trimN()

사용자가 지정한 개수의 노드를 힙에 삽입하거나 삭제합니다.

- growN()은 사용자가 지정한 N 개의 노드를 힙에 삽입합니다.
  - 힙 또는 CBT 에서 최대 키를 찾습니다.
  - 임의의 키를 저장할 키 타입의 배열을 할당합니다.
  - randomN()을 호출하여 [(max + 1)...(max + count)] 범위 내 임의의 키를 가져옵니다.
  - 함수를 호출하여 keys[]에 키를 한 번에 하나씩 삽입합니다.
  - 노드를 삽입할 때마다 DEBUG 가 정의된 경우 힙을 선택적으로 출력합니다.

- trimN()은 사용자가 지정한 N 개의 노드를 힙에서 삭제합니다.
  - 삭제하려는 노드의 개수가 힙의 크기보다 크면 노드의 개수를 힙의 크기로 설정합니다.
  - 노드 삭제에 사용할 함수 포인터를 설정합니다.

- 함수를 호출하여 노드를 하나씩 삭제합니다.
- 노드를 삭제할 때마다 DEBUG 가 정의된 경우 힙을 선택적으로 출력합니다.

Step 2.5: 알고리듬 개선 및 수정

heapOrdered() 함수는 힙 순서의 구조를 재귀를 사용하여 확인합니다. 다음 코드(heap.cpp)에는 실행 속도를 개선할 여지가 있습니다. 전체 코드를 새로 작성하거나 새 알고리듬을 사용하지 않고 개선해 보세요. 예를 들어, 반복을 사용하여 해결하지 마세요.

힌트: 불필요한 계산 또는 검사를 줄이세요.

```
/** is it heap-ordered at a node k? */
bool heapOrderedAt(heap p, int k) {
 if (k > p->N) return true;

 int left  = 2 * k;
 int right = 2 * k + 1;

 if ((left <= p->N) && p->comp(p, k, left))  return false;
 if ((right <= p->N) && p->comp(p, k, right)) return false;

 return heapOrderedAt(p, left) && heapOrderedAt(p, right);
}

// is it heap-ordered?
bool heapOrdered(heap p) {
 if (empty(p)) return false;
 return heapOrderedAt(p, 1);
}
```

과제 제출

- 소스 파일 상단에 아래와 같이 아너 코드 문장을 적고 서명하세요.

  On my honor, I pledge that I have neither received nor provided improper assistance in the completion of this assignment.

  서명: _____ 분반: _____ 학번: _____
- 제출하기 전에 코드가 제대로 컴파일이 되고 실행되는지 확인하세요. 제출 직전에 급하게 코드를 수정한 후 코드가 제대로 컴파일이 될 거라고 짐작하지 않는 게 좋습니다. "거의" 작동하는 코드도 틀린 것입니다.
- 과제가 컴파일 및 실행된다면, 마감 기한 전까지 과제의 일부만 완성했더라도 제출하기 바랍니다. 컴파일 및 실행되지 않는다면 제출하지 마세요. 마감 시간 이후 24 시간 이내 제출하면, 만점에서 25% 감점하고 채점합니다. 그 이상 늦은 것은 채점하지 않으며, 0 점 처리합니다.
- 제출 후, **마감 기한 전까지** 수정 및 재제출이 가능합니다. 파일 하나만 수정하더라도 해당 파일과 관련된 파일들을 모두 재제출해야 합니다. 재제출 횟수는 제한 없습니다. 마감 기한 전에 **가장 마지막으로** 제출된 파일을 채점할 것입니다.

제출 파일 목록

PSet Heapsort 는 다음 파일들을 제출합니다.

- Step 1.1: 다음 항목들을 포함한 하나의 파일을 제출하세요.
  1. 손으로 그린 그림의 이미지 캡처
  2. 힙 정렬 코드 실행 이미지 캡처 (heapsortx.exe 또는 본인의 실행 파일 사용)
  3. 질문과 답변
- Step 1.2: **heapsort.cpp**.

힙 & PQ(우선순위 큐)는 다음 파일들을 제출합니다.

- **heapprint.cpp & heap.cpp**

마감 기한 & 배점

- PSet 13: Heapsort
- PSet 14: Heap & PQ(Priority Queue)