**Binomial Heap 구현 및 실험**

**1. 서론**

Binomial Heap을 C로 구현하고 C++에서 제공하는 binary heap 알고리즘과 비교 실험을 진행하였다.

**2. 구현**

구현은 Introduction to Algorithm의 Binomial Heap 파트를 참고했다.

**2.1. bheap.{c,h}**

**#define TRUE (1==1)**

**#define FALSE (!TRUE)**

**#define SUCCESS (0)**

**#define FAILURE (1)**

에러 처리와 예외 값 처리를 위한 상수다.

**typedef int Key;**

원하는 데이터를 Key로 설정하여 사용할 수있다. 이번 실험에서 Key는 int로 설정하였다.

**typedef struct Node {**

**Key key;**

**int degree;**

**struct Node\* parent;**

**struct Node\* child;**

**struct Node\* sibling;**

**} Node;**

Binomial Heap을 구성하는 노드를 정의하는 구조체이다.

**// Empty heap is NULL pointer(0 Node).**

빈 힙은 노드가 하나도 없는 힙으로, Node\* 변수에 NULL을 할당하여 생성한다.

**#define GET(x, prop, none) ((x) ? (x)->prop : (none))**

**void print\_node(Node\* node);**

**void print\_tree(Node\* root);**

**void print\_roots(Node\* head, Node\* tail);**

**void print\_heap(Node\* heap);**

디버깅을 위해 정의한 시각화 함수이다. 각각 노드, binomial tree, root 리스트, heap을 출력한다.

**int link\_tree(Node\* parent, Node\* child);**

두개의 노드를 부모 - 자식 관계로 연결한다. merge\_heap에서 사용한다.

**Node\* merge\_roots(Node\* heap1, Node\* heap2);**

binomial heap 둘을 merge할 때, 그들의 루트 리스트를 O(n) 시간 복잡도로 합치기 위한 함수. 합쳐진 루트 리스트의 노드는 노드의 degree에 오름차순으로 정렬된다. 공간 복잡도는 O(1)이다.

**int merge\_heap(Node\* heap1, Node\* heap2, Node\*\* merged);**

binomial heap 둘을 merge한다. merge된 힙을 가리키는 포인터를 \*merged에 할당한다. Introduction to Algorithm에 제시된 알고리즘을 충실하게 구현하였다. 시간복잡도는 두 힙에 존재하는 노드의 합 N에 대해, 최악의 경우 O(log N)이다.

**int insert(Node\*\* heap, Node\* node);**

heap에 node를 삽입한다. node를 원소가 하나 있는 heap으로 간주하고 merge\_heap을 적용한다.

Key **min**(Node\* heap, Node\*\* min\_node);

heap에서 최소 원소를 찾아 key를 반환한다. Binomial heap은 min heap 속성을 따르므로, 모든 최소 원소는 root 리스트에 존재한다. Root 리스트에서 최소 원소를 찾아서 그 key를 반환한다.

Key **pop\_min**(Node\*\* heap);

heap에서 최소 원소를 찾아 제거하고 key를 반환한다. Binomial heap은 min heap 속성을 따르므로, 모든 최소 원소는 root 리스트에 존재한다. Root 리스트에서 최소 원소를 찾아서 삭제하고, 삭제한 root의 자식들로 degree의 오름차순으로 정리하여 새로운 heap을 만든다. 이를 기존 heap과 merge한다.

**int** **num\_tree\_node**(Node\* root);

**int** **is\_heap**(Node\* heap);

heap이 binomial heap이면 TRUE를 반환, 그렇지 않으면 FALSE를 반환한다. 1) heap에 존재하는 모든 binomial tree를 구성하는 노드 수가 2의 거듭제곱인지 확인한다. 2) heap에 존재하는 모든 binomial tree가 min heap property(부모 노드의 key < 자식 노드의 key)를 만족하는지 확인한다.

**2.2. unit test: test.cpp**

Binomial heap 구현을 위해 다양한 경우를 테스트하였다.

**TEST(link\_tree, test)**

**TEST(print\_tree, test)**

보조 함수 테스트

**TEST(merge\_roots, 1)**

**TEST(merge\_roots, 2)**

**TEST(merge\_roots, nil\_h1)**

**TEST(merge\_roots, property\_test)**

루트 리스트 머지 함수 테스트. property\_test의 경우 무작위 루트 리스트를 만들고 합친 뒤 정렬되었는지 확인.

**TEST(merge\_heap, merge\_2\_empty\_heap\_return\_empty\_heap)**

**TEST(merge\_heap, merge\_empty\_and\_1node\_return\_1node)**

빈 힙과 합치는 경우 테스트

**TEST(merge\_heap, case1\_just\_link\_roots)**

**TEST(merge\_heap, case2\_included)**

**TEST(merge\_heap, case3\_link\_two\_deg2\_heap\_to\_deg4\_heap)**

**TEST(merge\_heap, case4\_link\_two\_deg2\_heap\_to\_deg4\_heap)**

Introducton to Algorithm에 제시된 4가지 케이스에 대한 테스트

**TEST(heap, insert)**

**TEST(heap, pop\_min)**

**TEST(heap, min)**

**TEST(heap, prop\_test)**

각 operation에 대한 테스트. prop\_test는 무작위로 heap을 만들고 무작위로 opertion을 수행해서 heap이 is\_heap을 통과하는지 확인한다.

**3. 실험**

**3.1. 실험 구성**

구현한 binomial heap과 C++에서 제공하는 heap을 비교하기 위해 두가지 실험을 진행하였다. 첫번째 실험으로 push/pop 성능을 확인하기 위해, key를 오름차순/내림차순/무작위(중복 key 존재)로 삽입하고 삭제할 때 걸리는 시간을 측정했다. 두번째 실험으로는 힙에 존재하는 최대 key 수 N을 정하고, N 이하 무작위 개수의 key를 가지는 힙 h1,h2를 생성한다. 이 둘을 merge하는데 걸리는 시간을 측정한다.

**4. 실험 결과**

bh가 붙는 결과는 binomial heap을 사용한 결과, h가 붙은 결과는 C++에서 제공하는 (binary) heap알고리즘을 사용한 결과이다. push은 각 자료구조에 key를 삽입하는데 걸린 시간, pop은 최소 값을 삭제하는데 걸린 시간이다. asc는 key를 오름차순으로 삽입한 결과, dec는 내림차순, rand는 무작위 순서로 key를 삽입(중복 허용)한 결과이다.

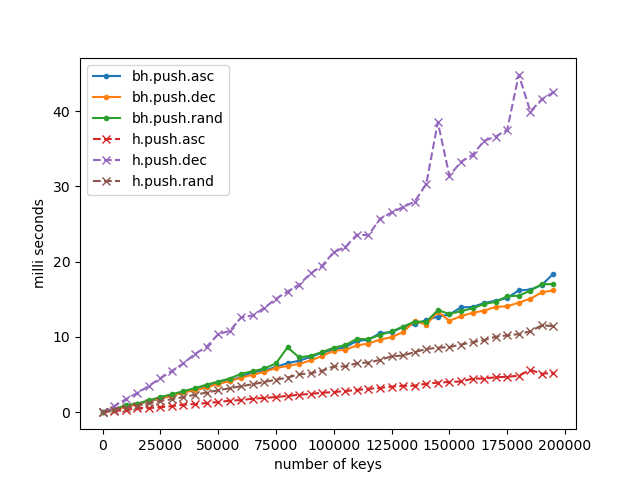


그림 1. 데이터에 대한 binomial heap, (binary) heap의 push 수행 시간

표 1. push수행시간 상승 비율. 수행시간을 외삽하여 기울기를 측정하였다.

|  |  |  |
| --- | --- | --- |
| 자료구조  데이터 | Binary Heap | Binomial Heap |
| 오름차순 | **2.680 \* 10-5** | 9.052 \* 10-5 |
| 내림차순 | 2.218 \* 10-4 | **8.443 \* 10-5** |
| 무작위 | **5.803 \* 10-5** | 8.833 \* 10-5 |

key를 내림차순으로 삽입한 경우를 제외하고는 binary heap이 binomial heap보다 수행시간이 짧게 걸렸다. 오름차순의 경우 이진힙을 만들 때 배열을 거의 변경하지 않아도 되기 때문에 가장 빠르다.

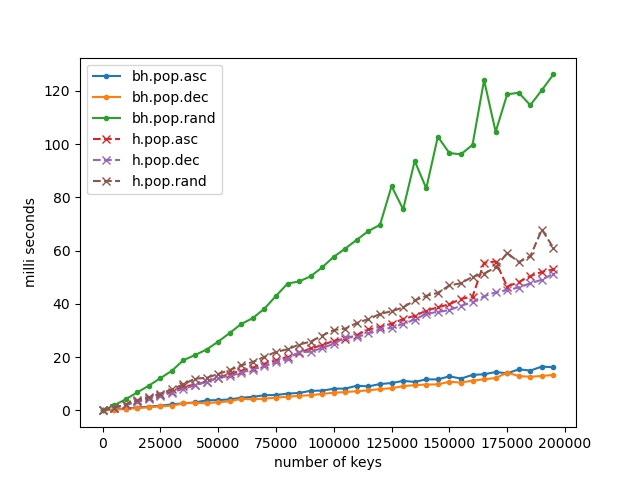


그림 2. 데이터에 대한 binomial heap, (binary) heap의 pop 수행 시간

표 2. pop 수행시간 상승 비율. 수행시간을 외삽하여 기울기를 측정하였다.

|  |  |  |
| --- | --- | --- |
| 자료구조  데이터 | Binary Heap | Binomial Heap |
| 오름차순 | 2.856 \* 10-4 | **8.422 \* 10-5** |
| 내림차순 | 2.629 \* 10-4 | **7.183 \* 10-5** |
| 무작위 | **3.273 \* 10-4** | 6.764 \* 10-4 |

무작위로 삽입하여 생성한 binomial heap에서 최소 원소를 pop하는 것을 제외 하고는 binary heap보다 더 나은 수행시간 성능을 보였다. Push와 Pop operation은 key를 어떻게 넣느냐에 따라서 두 자료구조의 성능이 달랐다.

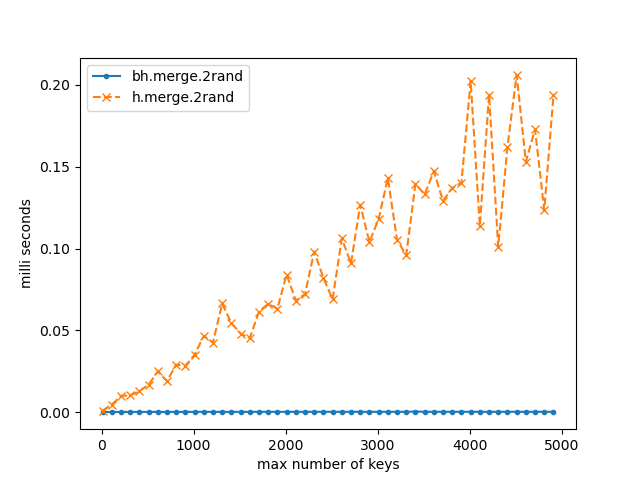


그림 3. 데이터에 대한 binomial heap, (binary) heap의 merge 수행 시간

표 3. merge 수행시간 상승 비율. 수행시간을 외삽하여 기울기를 측정하였다.

|  |  |  |
| --- | --- | --- |
| 자료구조  데이터 | Binary Heap | Binomial Heap |
| 무작위 | 3.658 \* 10-5 | **2.460 \* 10-8** |

merge의 경우 binomial heap이 binary heap보다 더 나은 성능을 보였다.

**5. 결론**

이번 과제에서는 binomial heap을 구현하고, C++에서 제공하는 heap 알고리즘과 성능을 비교해 보았다. pop과 push operation에서는 binomial heap과 C++의 heap 사이의 우열을 가리기 힘들다. 그러나 merge 에서는 Binomial heap이 C++의 heap보다 효율적이었다.