Binomial Heap Operations

Guru Nanak
IBM18CS031
Gaurang

Write-Up:

```
// A Binomial Tree node
Struct Node {
    int data, degree;
    Node *child, *sibling, *parent;
};

Node* newNode(int key){
    Node *temp = new Node;
    temp -> data = key;
    temp -> degree = 0;
    temp -> child = temp -> parent = temp -> sibling = NULL;
    return temp;
}

Node* mergeBinomialTrees(Node *b1, Node *b2){
    if (b1 -> data > b2 -> data)
        swap(b1, b2)

    b2 -> parent = b1;
    b2 -> sibling = b1 -> child
    b1 -> child = b2
    b1 -> degree ++;
    return b1;
}

list<Node *> unionOnBinomialHeap(list<Node*> l1, list<Node*> l2
{
    list<Node *> new;
    list <Node*>:: iterator it = l1.begin()
    list <Node*>:: iterator ot = l2.begin();
    while (it != l1.end() && ot != l2.end()){
        if ((*it) -> degree <= (*ot) -> degree){
            new.push_back(*it); it++;
        }
```

(1)

```
    else {
        _new.push_back(*ot); ot++;
    }
}
while (it != l1.end()) {
    _new.push_back(*it); it++;
}
while (ot != l2.end()) {
    _new.push_back(*ot); ot++;
}
return new;
}
list<Node *> adjust(list<Node *> _heap) {
    if (_heap.size() <= 1
        return _heap
    list<Node *> new_heap;
    list<Node *>::iterator it1, it2, it3;
    it1 = it2 = it3 = _heap.begin();
    if (_heap.size() == 2) {
        it2 = it1;
        it2++;
        it3 = _heap.end();
    }
    else {
        it2++;
        it3 = it2;
        it3++;
    }
    while (it1 != _heap.end()) {
        if (it2 == _heap.end())
            it1++;
        else if ((*it1)->degree < (*it2)->degree) {
            it1++; it2++;
            if (it3 != heap.end())
                it3++;
        else if (*it1->degree == *it2->degree) {
            Node *temp;
            *it1 = mergeBinomialTrees(*it1, *it2);
            it2 = _heap.erase(it2);
            if (it3 != _heap.end())
                it3++;
        }
    }
    return heap;
```

```
list<Node*> insertATreeInHeap(list<Node*> _heap, Node *tree){
    list<Node*> temp;
    temp.push_back(tree);
    temp = unionBinomialHeap(_heap, temp);
    return adjust(tree);
}
list<Node*> insert(list<Node*> head, int key){
    Node *temp = newNode(key);
    return insertATreeInHeap(_head, temp);
}
Node* getMin(list<Node*> _heap){
    list<Node*>::iterator it = _heap.begin();
    Node *temp = *it;
    while(it != heap.end()){
        if((*it)->data < temp->data)
            temp = *it
        it++;
    }
    return temp;
}
list<Node*> removeMinFromTreeReturnBHeap(Node *tree){
    list<Node*> heap;
    Node *temp = tree->child;
    Node *lo;
    while(temp){
        lo = temp;
        temp = temp->sibling;
        lo->sibling = NULL;
        heap.push_front(lo);
    }
    return heap;
}
```

(3)

```
list<Node*> extractMin(list<Node*> _heap){
    list<Node*> new_heap, lo;
    Node *temp;

    temp = getMin(_heap);
    list<Node*>::iterator it;
    it = _heap.begin();
    while(it != _heap.end()) {
        if(*it != temp)
            new_heap.push_back(*it);
        it++;
    }
    lo = removeMinFromTreeReturnBHeap(temp);
    new_heap = unionBinomialHeap(new_heap, lo);
    new_heap = adjust(new_heap);
    return new_heap;
}
void printTree(Node *h) {
    while(h) {
        cout << h->data << " ";
        printTree(h->child);
        h = h->sibling;
    }
}
void printHeap(list<Node*> _heap) {
    list<Node*>::iterator it;
    it = _heap.begin();
    while(it != _heap.end()) {
        printTree(*it);
        it++;
    }
}
```