Implementation of binomial heap

Ankitha
9/12/20

```
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;
}
```

For merging two binomial trees we write following function.

```
Node * mergeBinomial(Node *b1, Node *b2)
{
    if (b1->data > b2->data) // child of b1 is smaller
        swap(b1, b2);

    // make a larger value tree as child of smaller
    b2->parent = b1;
    b2->sibling = b1->child;
    b1->child = b2;
    b1->degree++;
    return b1;
}
```

For performing union on 2 binomial heap.

```
list<Node> unionBinHeap(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++;
        }
```

```
else
{
    - new.push_back(*ot);
    ot++;
}
// now we need to check if there are few elements
left out in either of heaps
while(it1 != l1.end())          // for l1
{
    - new.push_back(*it1); it1++; }
while (ot != l2.end())
{
    - new.push_back(*ot); ot++; }
return new;
}
```

Following is for inserting a binomial tree into the heap

```
list<Node*> insertTree(list<Node*> - heap, Node *tree)
{
    list<Node*> temp; // new heap
    temp.push_back(tree);
    temp = unionBinomialHeap(-heap, temp);
    return adjust(temp);
}
```

Removing min key element from the heap.

```
list<Node*> removemin(Node *tree)
{
    list<Node*> heap;
    Node *temp = tree->child;
    Node *lo;
    // making a binomial heap from the tree
    while(temp)
    {
        lo = temp;
        temp = temp->sibling;
        lo->sibling = NULL;
        heap.push_front(lo);
    }
    return heap;
}
```

```
// inserting a element
list <Node*> insert(list<Node*> _head, int key)
{
    Node *temp = newNode(key);
    return Insertion(_head, temp);
}
// return a pointer to minimum value
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*> extractMin(list <Node*> _heap)
{
    list
    temp = getMin(_heap);
    it = _heap.begin();
    while(it != _heap.end())
    {   if (*it != temp)
        {   into new_heap.push_back(*it);
        }
        it++;
    }
    lo = removeMin(temp)
    new_heap = unionHeap(new_heap, lo);
    return new_heap;
}
```