A New Algorithm for the Josephus Problem Using Binary Count Tree

Jian Li[a], Yanzhou Ma[b], Zesheng Gao[c] and Xinyu Hu[c]
Luoyang Campus of the PLA Information Engineering University, Luoyang, China
[a]maomaotfntfn@163.com, [b]myz827@126.com, [c]huxinyu1002015@163.com

Abstract. This paper presents a new efficient algorithm for the Josephus problem using Binary Count Tree. Binary count tree is a data structure ameliorated from the BST which additionally records the node-count of each sub tree. In this algorithm, the whole Joseph circle is put into the binary count tree. The target node can be quickly found by moving the pointer between the child node and the parent node, and then the target will be deleted and the related node-counts will be updated. Analysis and experiments shows that this algorithm can find out an entire Joseph sequence in time O(n*log n).

Keywords: Josephus problem, BST, Binary count tree, Binary recursion.

1. Introduction

Josephus Problem is a classic mathematics and computer problem. It can be described as follows: there are n people consecutively numbered from 1 through n and arranged in a circle. They are forced to count off. And he who counts digit m off is out and the one next to him will count 1 again, next person counting m likewise. In this way, people in the circle become fewer and fewer and the last becomes the winner. There are many other versions of the story, and they will not be narrated here.

Via the description above, two questions of different layers can be posed.

Question 1. If the last winner is our only concern, then it can be described as:

Given positive integers n and m, then find out the number of winner - W(n, m).

For example, W (10, 3) = 4.

Question 2. If each person's position in the killing sequence is wanted, it can be described as:

Given positive integers n and m, determine the entire sequence - List (n, m).

For instance, List (10,3) = {3,6,9,2,7,1,8,5,10,4}.

It can be easily found that Q2 is more complicated, for the reason that the answer of Q1 is included in Q2. Q1 an be determined in time O(n) using recursion and we've already got a mature algorithm[1], which will not be introduced in detail here. This paper is meant to design a more concise and efficient method to solve Q2 in which an improved binary tree is used and then verify it.

2. Binary Count Tree

Binary Search Tree (BST for short) also named as Sorted Binary Tree, is a simple and efficient data structure usually used to sort and find data rapidly. Its basic features are as follows[2]:

(1) Every node in BST has two children (called the left child and the right child) at most;

(2) Every node value is greater than any value in its left subtree but less than the other side.

Binary Count Tree (BCT for short) is a data structure improved from the BST, which still has BST's basic features and extra records the node count of each subtree (shown as Fig.1).

It has two more domains than the Binary tree, pointing to the pointer of parent and node number of its subtree separately. In C language, node structure of BCT can be defined as followed:

```
typedef struct BCTree
{
int data;              // data area (also key)
Fig BCTree *L;                 // left child
Fig BCTree *R;         // right child
Fig BCTree *P;         // parent
int count;          // count of sub tree
} TNode;
```

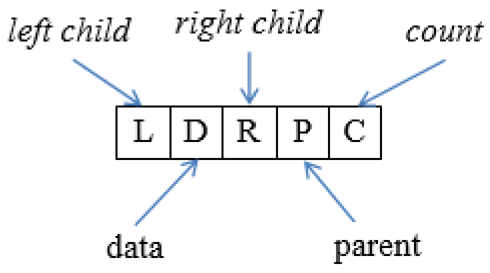


Fig .1The node structure of BCT

In the above code, data denotes the data domain(also the keyword); count denotes node count of its subtree; L,R and P points to L-child, R-child and parent respectively. The nodes structure of a BCT is shown as follows:

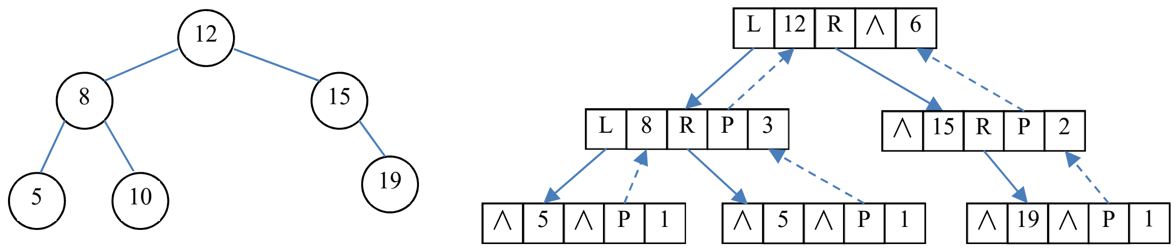


Fig. 2 The nodes structure of a BCT

Basic operations of the BCT include generating a tree, adding elements, deleting elements, going forward k steps, etc. Some of the operations will be described in detail in the following parts.

3. Algorithm Theory and Implementation

In the simple algorithm version of determining Josephus (n, m), we usually use a circular linked list to represent the Josephus ring[3-4], and the time needed to find the next target element is O(m). If m is very close to n, then the total time will be $O(n^2)$.

The fact that the pointer in the circular linked list can only move forward one step at a time leads to low efficiency. However, if each movement can span multiple nodes, then the algorithm can be improved by reducing the total moves, and using BCT provides the possibility.

3.1 Create a BCT

Original Josephus ring is made up of n elements. These elements can be used as nodes to generate a "binary counting tree" via binary recursion. Take the sequence[1:14] as an example, the whole sequence is divided into two parts by the intermediate element 7. Then we use it as the root, and make the two parts as the left and right subtrees(shown as fig.3).

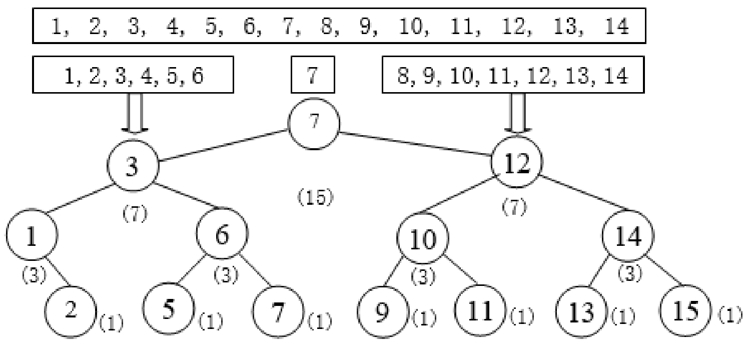


Fig. 3 The process of creating a BCT

Due to the binary recursion we used, the size of the left and right subtrees at each node is almost the same(with difference no more than 1), and the binary tree generated is nearly full. The height of the tree generated by n nodes is h, so we can figure out its expression:

$$h = \lfloor log_2 n \rfloor + 1$$

The C language code for the generation of BCT is as follows:

```
TNode * CreateBCT( int low, int high, TNode * Parent)
{   if (low <= high)          //if the sequence is not empty
{      int mid = (low + high) / 2;
TNode * M = (TNode *) malloc (sizeof(TNode));    M->data = mid;    M->count = high-low+1;
M->L = CreateBCT (low, mid-1, M);    M->R = CreateBCT (mid+1, high, M);
M->P = Parent;    return M;
}
return NULL;        //if the sequence is empty
}
```

In the code above, the starting and ending position of the sequence are represented by parameter low and high, and the parameter parent will be the parent of the intermediate elements. To generate the "binary counting tree", we should allocate the node space for intermediate element M, set its data fields (data), and node counter (count) and parent pointers. Recursion produces the left and right subtrees with intermediate nodes as their parents. The parameter parent can be set to NULL when we first call the function, because the root of the entire tree has no parent nodes. The algorithm above can be completed with O(n) time.

3.2 Take k Step Forward

Using the BCT to Solving Josephus(n, m), the problem of counting off is converted into: how to quickly find the k-th node after the current node (marked as T) in the tree (where k=m- 1).

In order to describe it conveniently, we use Step (T, k) to represent the task of "taking k step forward from T". T′ and k′ are used to represent the T after moving and the updated k respectively.

In Step (T, k), k needs to go forward when its value is positive and go backward otherwise. When its value equals to 0,the task is finished. Step (T, k) will be transformed into Step(T', k') after one step, so it's clear that our aim is to turn k into 0 through the moment of T.

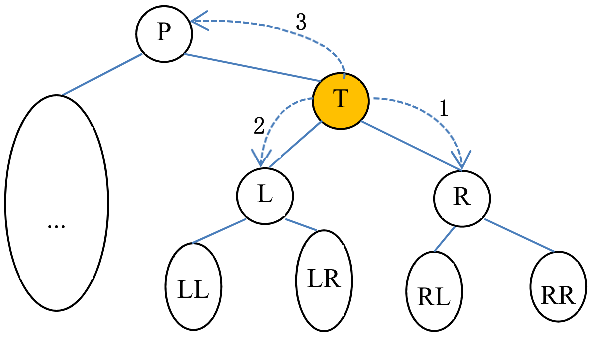


Fig. 4 The discussion of the node's moving direction

If k !=0,we need to move the current node and possible movement directions include: to the left child, to the right one or the parent. Three different cases are discussed as follows:

Case 1: If 0< k <= Count (R), the target node must be in the left subtree: make T ′= L,k′ = k+ (Count (LR)+1).

Case 2: If 0> k >= -Count (L), the target node must be in the right subtree: make T ′= L,k′ = k+ (Count (LR)+1).

Case 3: If k > Count (R) or k<-Count (L), there are two cases to discuss:

Case 3.1: If T has parent node P, then T'= P; If T is the left child of P, k′ = k - (Count (R) + 1); If T is the right child, k′ = k + (Count (L) + 1).

Case 3.2: If T doesn't has parent, then it is the root. Take n as the sum of nodes in the tree, we have: n = Count (R) + Count (L) + 1

If k> Count (R), then move forward a circle (n steps), and let T'= T, k' = k-n; we can proves that k′ = k - n> - (Count (L) + 1) >= -Count (L). We will fall in "Case 2" at the next time we search.

If k< -Count (L), go backward a circle (n steps), so that T '= T, k' = k + n; we can proves that k '= k + n < Count (R) + 1 <= Count (R). The next time we will fall in "Case 1".

The code for implementing the task of taking k steps forward is as follows:

```
TNode* Step (TNode *T, int k)
{if(k==0) return T;     // T just is the target
      int Rc, Lc, RLc, LRc;    GetCounts (T, &Rc, &Lc, & RLc, &LRc);
  if( 0<k && k<=Rc)      //Turn to the right subtree (0<k<=Rc)
  {    T=T->R;      k =k -(RLc+1);    }
  else if(-Lc<=k && k<0)   //Turn to the left subtree (-Lc<=k<0)
  {    T=T->L;      k = k + (LRc+1);    }
  else                    //Turn to the root node
  {    if(T->P != NULL) //If it has a parent
       {    if(T->P->L==T)  //If it’s the left child of the parent
            {     T = T->P;      k = k - ( Rc + 1); }
            else        //If it’s the right child of the parent
            {     T = T->P;      k = k + ( Lc + 1);}
       }
       else              //If it doesn’t have a parent (T is the root)
       {    if(k>0)         //If it needs to move forward
            {    k = k - T->count;      }      //Move forward a circle
            else if(k<0)    //If it needs to go backward
            {    k = k + T->count;      }       //Go backward a circle
       }
  }
  return    Step( T, k);
}
```

3.3 Delete the Target Node

It’s necessary to delete the target node when we find it, which corresponds to the element dequeue in Josephus ring. As for the deletion of a given node (T), it can be divided into the following categories:

Case 1: If T is the leaf, delete it directly.

Case 2: If T only has one child(shown in fig.5-a), delete T and replace its previous position with the child of T .

Case 3: If T has two children(shown in fig.5-b), find S, the direct of T, replace the data domain of T with S’s, and then delete S. It definitely apply to Case 1 and 2, due to the fact that S is the leftmost mode in the right subtree and has only one child at most.

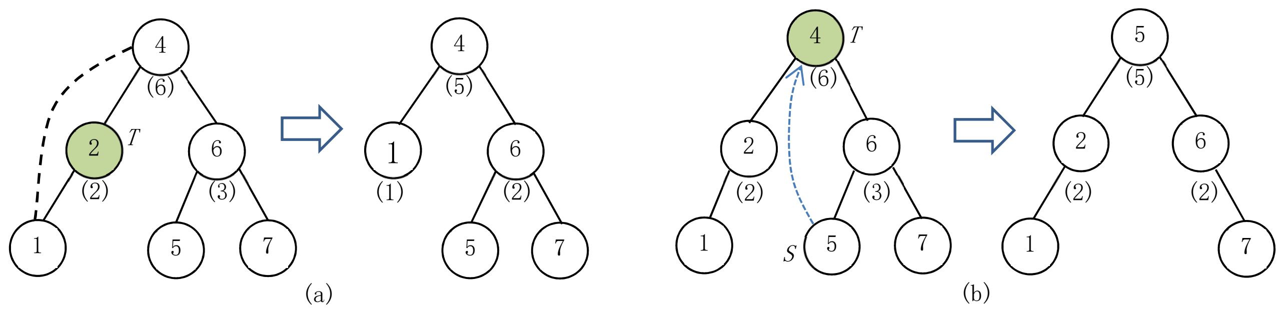


Fig. 5 Delete a node from the BCT

Remaining nodes need to be updated after deletion: no matter which case it is, subtract 1 from all ancestors of the node which is actually deleted. The delete operation is done in the RemoveTNode() function and the next node is returned. The space is limited and the code is omitted.

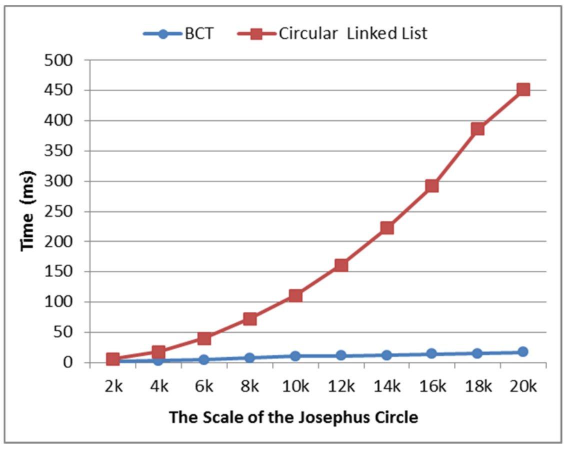
4. Tests and Analysis

Key steps have been discussed in the last section, and it will be easy for us to figure out the whole Josephus permutation by using them. The main algorithm code is as follows:

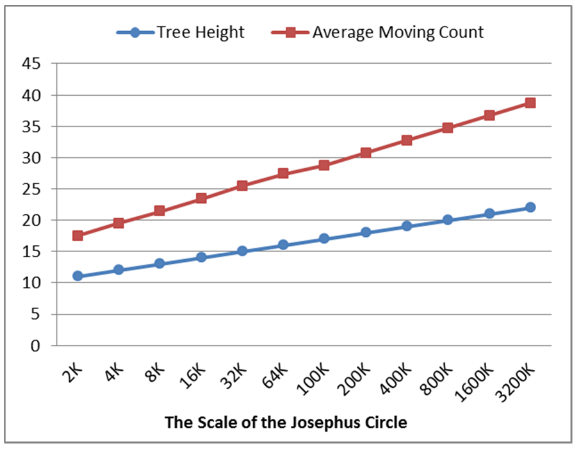
```
int *Josephus (int n, int m)
{       int * list = (int *) malloc (sizeof (int)*n);
        TNode *root = CreateBCT (1, n, NULL);
TNode *T = GetFirst (root);
        for (int rest =n; rest>0; rest--)
        {     int k = (m-1) % rest;
T = Step (T, k);      //take k steps forward
list[i] = T->data;
T = RemoveTNode (T);      //delete the target node
        }
return list;
}
```

In the algorithm illustrated above, O(n) time is used to execute CreateBCT() , and GetFirst() O(log n) time ; a single execution of Step() and RemoveTNode() requires only the same time of O(log n); all in all, the time complexity of the whole algorithm is O(n log n).

In order to verify the correctness and efficiency of the algorithm, the new algorithm presented in this paper is compared with a simple algorithm based on circular linked list. Test programs were developed in VS2010 and run on a computer with 8G RAM. Accounting the worst case, we conducted a time performance test on Josephus (n, n). The results show that the Josephus sequences obtained by the two algorithms are exactly the same; the simple algorithm shows a quadratic curve with the increase of problem scale , and the new one looks almost linear (shown as Fig.6-a).



(a)



(b)

Fig. 6 Statistics of experiment results

For the purpose of further verifying the time complexity of it, we expand the test scale and gather more statistics, including the height of the binary counting tree and the average number of movements (shown as Fig.6-b). The following test data shows that the increase of average moves related to the height of tree is linear, and the height of the tree is approximately equal to log(n). Therefore, we can figure out that the complexity of whole algorithm is O(n log n).

5. Summary

All the theorems and experiments mentioned above have proven that the algorithm put forward in this paper for the Josephus problem,which is based on BCT, is correct. And its time complexity is O(n log n). In the algorithm,the “counting information” additionally recorded in the BCT is made use of to find target node rapidly. Therefore, the maintenance of counting information is really crucial when an element is deleted. And if the equilibrium of BCT is taken into account,we can further enhance the efficiency of this algorithm. BCT is an improved data structure and can be used to solve many other algorithm problems except for the Josephus problem.

References

[1]. Information on http://rosettacode.org/wiki/Josephus_problem.

[2]. Bronson, N. G., Casper, J., Chafi, H., & Olukotun, K. (2010). A practical concurrent binary search tree. (Vol.45, pp.257-268). ACM.

[3]. Ruskey, F., & Williams, A. (2010). The Feline Josephus Problem. International Conference on Fun with Algorithms (Vol.50, pp.343-354). Springer, Berlin, Heidelberg.

[4]. Wang, Yong Hong. Comparison between Algorithms of Josephus Problem. Modern Computer (2008).