Data Structures and Objects
CSIS 3700

Fall Semester 2024 — CRN 41145

Project 4 — Treap Dictionary
Due date: Wednesday, December 11, 2024

Goal

Develop and implement a dictionary that uses treaps to hold keys and also uses shared data pools.

Details

This project will create a templated Dictionary class where the key and value types are placeholders in the template. In addition, the project has two interesting features:

- The keys will be stored in a *treap*. This will allow for key searches to be performed in $\lg n$ time, on average. This will allow for a sorted dictionary with better performance than the version discussed earlier in the semester.
- The dictionary will use a shared data pool. Data will be stored in parallel arrays, rather than individually allocated nodes, and two dictionaries with the same key and value types will share a common set of arrays. This provides better performance than individually allocating and deleting nodes.

The class will implement the following actions:

- The five basic actions — create, destroy, clear, size (both count and height) and isEmpty. These are almost identical to their equivalent actions in a binary search tree.
- `ValueType &search(const KeyType &k)`
  Search for key *k* in the dictionary. If it is there, return a reference to the corresponding value. If it is not there, throw a `domain_error` exception.
- `ValueType &operator[](const KeyType &k)`
  This is like `search()`, but if the key does not exist, it is added to the tree. This combines the insert and update behaviors of a conventional dictionary.
- `void remove(const KeyType &k)`
  This removes the key and its value from the dictionary. If the key does not exist, throw a `domain_error` exception.

▸*Shared data pools*

Instead of individually allocating and deleting nodes, we will allocate and delete blocks of "nodes", which are actually entries in a set of parallel arrays. Rather than having one node with a datum, left and right pointers, a count and a height, we will use a set of arrays, each holding one of the six values (the data have both keys and values). The arrays are allocated dynamically, so that array doubling can be employed when space runs out.

A shared pool means that all dictionaries with the same key and value types use the same parallel arrays for their allocation and deallocation. This is similar to how disk space is managed in FAT filesystems; all files get blocks of space from a shared list of available blocks.

Sharing creates some additional complexity in writing the code. However, it is more efficient than individual allocations.

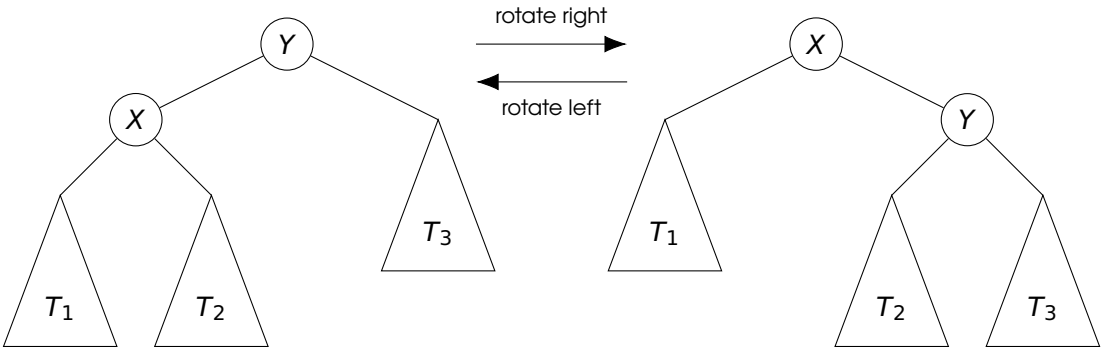
▹*Treaps*

A *treap* is a combination of a binary search tree and a max-heap. The keys are always maintained as a binary search tree, such that for any node n, descendant nodes with smaller keys are in the left subtree of n and any descendants with larger keys are in the right subtree.

Each node is assigned a random number when it is inserted into the treap. The treap is then arranged so that for each node n in the treap, the number in node n is larger than the number in either child of n. Thus, while a treap will almost certainly not have a heap shape (all levels except the bottom being completely filled and nodes on the bottom level as far left as possible), nodes in a treap do satisfy a max-heap relation (parent larger than children).

The heap relationship property is maintained by the use of *rotations*. A rotation reshapes a (sub)tree by shifting the relative position of two nodes — always a parent and a child — and their three subtrees (which may be empty.) The nodes can be rotated "right", where a left child replaces a parent node and the parent becomes the right child; they can also rotate "left" where a right child replaces its parent and the parent becomes the left child.

The following diagram illustrates left and right rotation.



Note that rotation does not affect the binary search tree property.

Rotations are used when inserting and removing nodes from a treap. When inserting into a treap, rotations are used to restore the parent-child heap relationships without damaging the binary search tree property. This has a randomizing effect on the overall treap shape in an attempt to maintain a $\Theta(\lg n)$ treap height. When removing a node, rotations are used to push the deleted node toward the bottom of the treap, which will eventually simplify the removal process.

▸*Setting up the class*

To set up the class, start by setting up two constants and populating the class with empty functions:

```
#ifndef _TREAP_DICTIONARY_H
#define _TREAP_DICTIONARY_H

#include <cstdint>      // for uint32_t
#include <stdexcept>    // for domain_error
#include <random>       // for random number generation

static const uint32_t
    NULL_INDEX = 0xffffffff,
    DEFAULT_INITIAL_CAPACITY = 16;

template <typename KeyType,typename ValueType>
class TreapDictionary {
public:
    explicit TreapDictionary(uint32_t _cap = DEFAULT_INITIAL_CAPACITY) {

    }

    ~TreapDictionary() {

    }

    void clear() {

    }

    uint32_t size() {

    }

    uint32_t height() {

    }

    bool isEmpty() {

    }

    ValueType &search(const KeyType &k) {

    }

    ValueType &operator[](const KeyType &k) {

    }

    void remove(const KeyType &k) {

    }

private:
```

```
52      uint32_t prvAllocate() {
53
54      }
55
56      void prvFree(uint32_t n) {
57
58      }
59
60      void prvClear(uint32_t r) {
61
62      }
63
64      void prvAdjust(uint32_t r) {
65
66      }
67
68      uint32_t prvRotateLeft(uint32_t r) {
69
70      }
71
72      uint32_t prvRotateRight(uint32_t r) {
73
74      }
75
76      uint32_t prvInsert(uint32_t r,uint32_t &n,const KeyType &k) {
77
78      }
79
80      uint32_t prvRemove(uint32_t r,uint32_t &ntbd,const KeyType &k) {
81
82      }
83  };
84
85  #endif
```

The public functions carry out the actions described at the top of the Details section. The private functions either assist the public functions or separate specific critical tasks into their own functions; they are private to hide implementation details from the end user.

The purpose of each private function follows.

- `uint32_t prvAllocate()`
  Allocate one node by selecting one unused node from the shared pool and marking it as in use. Returns the index of the allocated node. If all nodes in the pool are in use, the function performs array doubling to increase the pool size.
- `void prvFree(uint32_t n)`
  Delete one node by marking it as unused. Unused nodes are kept in a *free list* which is described later in this document.
- `void prvClear(uint32_t r)`
  This performs the recursive part of a postorder traversal to delete all of the nodes in the tree. This is similar to the process in a "normal" binary search tree.

- `void prvAdjust(uint32_t r)`
  This recalculates the node count and height of the (sub)tree with node r as the root.
- `void prvRotateLeft(uint32_t r)`
  This rotates the tree rooted at `r` to the left, and returns the new root (the right child of `r`.)
- `void prvRotateRight(uint32_t r)`
  This rotates the tree rooted at `r` to the right, and returns the new root (the left child of `r`.)
- `uint32_t prvInsert(uint32_t r,uint32_t &n,const KeyType &k)`
  This combines the insert and update actions from a conventional dictionary. Parameter n contains the index of a newly allocated node. If the given key does not exist in the tree, eventually node n will be used and returned. If the given key does exist, when it is found, n will be overwritten with the index of the key.
- `uint32_t prvRemove(uint32_t r,uint32_t &ntbd,const KeyType &k)`
  This performs the recursive removal of a node in a manner similar to a normal BST.

▸*Adding the data*

As with a conventional BST, the only value kept in the object is the index of the tree's root. However, there are twelve values shared across all dictionaries with the same key and value types. The shared values require some additional coding to initialize them; since they are shared, they cannot be initialized in the class constructor, as the constructor isn't invoked until an object is created.

Begin by adding the following to the private region of the class:

```
1     uint32_t
2         root;           // tree root
3
4     static uint32_t     // this is the shared data
5         *counts,        // counts for each node
6         *heights,       // heights for each node
7         *left,          // left node indexes
8         *right,         // right node indexes
9         *heapVals,      // random values for the treap
10        nTreaps,        // number of treaps with this key and value type
11        capacity,       // size of the arrays
12        freeListHead;   // the head of the unused node list (the free list)
13
14    static std::random_device
15        *rd;            // source of random noise
16    static std::mt19937
17        *mt;            // a random number generator
18    static KeyType
19        *keys;          // pool of keys
20    static ValueType
21        *values;        // pool of values
```

Initializing the static members is done outside of the class. The initialization for `counts` looks like this:

```
1  template <typename KeyType,typename ValueType>
2  uint32_t *TreapDictionary<KeyType,ValueType>::counts = nullptr;
```

The other eleven shared values are initialized in a similar manner. All of the pointer values should be set to `nullptr` and the three integers should be set to 0. All initialization for static data is done after the class, before the `#endif` line.

▹*Algorithms*

Note: `clear()`, `size()`, `height()`, `isEmpty()`, `remove()` and `prvClear()` are virtually identical to their regular BST counterparts, so they will not be shown here. The only differences are that indexes are used instead of pointers, parallel arrays are used instead of node structures, and `prvFree()` is used to delete a node in the `prvClear()` function.

Constructor

The constructor initializes the tree root as usual. However, if this is the first dictionary with the given key and value types, then it must also allocate space for the shared data pool.

Algorithm 1 Initializing a treap dictionary

Preconditions None

Postconditions The shared data pool has been created
The root is initialized

```
1:  procedure TreapDictionary(cap = INITIAL_DEFAULT_CAPACITY)
2:      if nTreaps = 0 then
3:          Allocate space for the seven arrays          ▹ cap is the initial size of all arrays
4:          capacity ← cap
5:          Generate the free list
6:          Dynamically allocate the random number generator objects
7:      end if
8:      nTreaps ← nTreaps + 1
9:      root ← NULL_INDEX
10: end procedure
```

Generating the free list

Note: This is not a standalone function; this is part of the constructor and `prvAllocate()` functions. Do not make it a function.

Generating the free list is done by connecting all of the unused nodes into a basic linked list. The process is easy, since all of the unused nodes will be in a contiguous range of nodes — all unused nodes are all adjacent to each other. The process is the same in both the constructor and `prvAllocate()`; all that changes are the start and end indexes. In the constructor, the start index is 0 and the end index is $cap - 1$. In `prvAllocate()`, the start index is $capacity$ and the end index is $2 \cdot capacity - 1$.

Algorithm 2 Generating the free list

Preconditions Shared data arrays were either just created or extended

Postconditions The unused nodes are connected into a linked list
freeListHead is the index of the first unused node

```
1: procedure GenerateFreeList(start,end)
2:     for i ← start to end − 1 do
3:         left[i] ← i + 1
4:     end for

5:     left[end] ← NULL_INDEX

6:     freeListHead ← start
7: end procedure
```

Destructor

The destructor is almost the same as it is for a regular BST. The difference is that if the last dictionary is being destroyed, then instead of using `prvClear()` to delete the nodes, the entire shared data space is deleted.

Algorithm 3 Destructor

Preconditions None

Postconditions If other dictionaries exist, nodes in this dictionary are deallocated
Otherwise, shared space is deleted

```
1: procedure ~TreapDictionary
2:     nTreaps ← nTreaps − 1

3:     if nTreaps = 0 then
4:         Delete all shared arrays
5:
6:         Delete random number objects
7:     else
8:         prvClear(root)
9:     end if
10: end procedure
```

Search

Searching is similar to searching a regular BST. The main difference is that here a reference to the key's corresponding value is returned. The value is returned by reference because (a) it may be large, so space is saved and (b) it allows for the possibility of using the result to assign a new value to the key.

An exception is thrown if the key is not in the dictionary.

Algorithm 4 Searching for a key

Preconditions k is a key that may or may not exist in the dictionary

Postconditions A reference to the key's value is returned if the key exists in the dictionary
A `domain_error` is thrown if the key is not in the dictionary

```
1:  procedure SEARCH(k)
2:      n ← root
3:      while n ≠ NULL_INDEX do
4:          if k = keys[n] then
5:              return values[n]
6:          else if k < keys[n] then
7:              n ← left[n]
8:          else
9:              n ← right[n]
10:         end if
11:     end while

12:     throw domain_error("Search: Key not found")
13: end procedure
```

Insert / update / access

The `operator[]` function provides a combined insert, update and access operation. It is based on the conventional BST insert.

Algorithm 5 Accessing a key's value

Preconditions k is a key that may or may not already exist in the dictionary

Postconditions If k is not in the dictionary, it is inserted into the dictionary
The function returns a reference to k's corresponding value

```
1:  procedure OPERATOR[](k)
2:      tmp ← PRVALLOCATE()

3:      n ← tmp

4:      root ←PRVINSERT(root, n, k)

5:      if n ≠ tmp then
6:          PRVFREE(tmp)                    ▷ Deallocate unused node
7:      end if

8:      return values[n]
9:  end procedure
```

Allocating a node

We are managing the shared data pool; therefore, we must also manage allocation and deallocation of individual nodes.

The process is simple; take the first node out of the free list and use it. There is one catch: if the free list is empty, then we must allocate a larger data pool and build a new free list from the extra nodes.

Algorithm 6 Allocating a node

Preconditions None

Postconditions The index of an unused node is returned

```
1:  procedure PRVALLOCATE
2:      if freeListHead = NULL_INDEX then
3:          Allocate temporary arrays with 2 · capacity elements
4:          Copy data from old arrays to temporary arrays
5:          Delete old arrays
6:          Point shared pointers to temporary arrays
7:          Regenerate the free list
8:          capacity ← 2 · capacity
9:      end if
10:     tmp ← freeListHead
11:     freeListHead ← left[freeListHead]
12:     left[tmp] ← NULL_INDEX
13:     right[tmp] ← NULL_INDEX
14:     counts[tmp] ← 1
15:     heights[tmp] ← 1
16:     heapVals[tmp] ← random number          ▷ Use (*mt)() to generate a random number
17:     return tmp
18: end procedure
```

Deallocating a node

Deallocation is very simple — just add the node to the front of the free list.

Algorithm 7 Deallocating a node

Preconditions n is a node to be deleted

Postconditions n is added to the front of the free list

```
1: procedure PRVFREE(n)
2:     left[n] ← freeListHead

3:     freeListHead ← n
4: end procedure
```

Adjusting height and count information

Note that GETCOUNT() and GETHEIGHT() can be either separate functions or macros.

After insertion or removal of a node, the count and height information for all of the node's ancestors must be recomputed.

Algorithm 8 Adjusting height and count information

Preconditions r is a node whose count / height information may have changed

Postconditions Height and count information for r are correct

```
1: procedure PRVADJUST(r)
2:     counts[r] ←GETCOUNT(left[r])+GETCOUNT(right[r]) + 1

3:     heights[r] ← max(GETHEIGHT(left[r]),GETHEIGHT(right[r])) + 1
4: end procedure
```

Recursively inserting / updating / accessing a node

This process is similar to recursively inserting into a regular BST. There are two main differences:

- If the key already exists, its index is used and no new node is inserted. The node allocated in the OPERATOR[ ] procedure will be deallocated in that case.
- When backing out of recursion, the random heap values of the root and returned value are compared. If the child is larger, the tree is rotated to make the child the new root of this subtree.

Algorithm 9 Recursive insert / update / access

Preconditions r is the root of the tree that should contain k
n is the index of an freshly allocated node
k is a key that may or may not exist in the treap

Postconditions k is in the tree
n is the index of the node containing k

```
1:  procedure PRVINSERT(r,n,k)
2:      if r = NULL_INDEX then                          ▷ k not in tree, insert it here
3:          keys[n] ← k
4:          return n                                    ▷ n is root of formerly empty tree
5:      end if
6:      if k = keys[r] then
7:          n = r                                       ▷ key found, remember where
8:      else if k < keys[r] then
9:          left[r] ←PRVINSERT(left[r], n, k)
10:         if heapVals[r] < heapVals[left[r]] then
11:             r ←ROTATERIGHT(r)
12:             PRVADJUST(right[r])
13:         end if
14:     else
15:         right[r] ←PRVINSERT(right[r], n, k)
16:         if heapVals[r] < heapVals[right[r]] then
17:             r ←ROTATELEFT(r)
18:             PRVADJUST(left[r])
19:         end if
20:     end if
21:     PRVADJUST(r)
22:     return r
23: end procedure
```

Recursively removing a node

Removal from a treap is much simpler than removing a node from a regular binary search tree. In a BST, an attempt is made to minimize alterations to the tree. However, self-adjusting trees have no such limitation.

In a conventional BST, most removal cases are simple; the only complex case is removing a node that has two children. Treaps simplify this case by rotating the node to be deleted downward. Eventually, this node will no longer have two children and will be easy to remove.

This is the algorithm to recursively remove of a node from a treap.

Algorithm 10 Recursively removing a key

Preconditions r is the index of the root of a (sub)tree that may contain k
k will not be in any other subtree

Postconditions $ntbd$ holds the index of the node to be deleted, if it exists in the tree
r is the root of the tree after removing $ntbd$

```
1:  procedure PRVREMOVE(r,ntbd,k)
2:      if r = NULL_INDEX then                          ▷ Subtree is empty, k is not in tree
3:          throw domain_error("Remove: Key not found")
4:      end if

5:      if k < keys[r] then                             ▷ k might be in left subtree
6:          left[r] ←PRVREMOVE(left[r], ntbd, k)
7:      else if k > keys[r] then                        ▷ k might be in right subtree
8:          right[r] ←PRVREMOVE(right[r], ntbd, k)
9:      else                                            ▷ k is in node r
10:         ntbd ← r

11:         if left[r] = NULL_INDEX then
12:             if right[r] = NULL_INDEX then           ▷ r is a leaf, subtree is removed
13:                 r ← NULL_INDEX
14:             else                                    ▷ r only has right child, it is new root
15:                 r ← right[r]
16:             end if
17:         else
18:             if right[r] = NULL_INDEX then           ▷ r only has left child, it is new root
19:                 r ← left[r]
20:             else                                    ▷ r has two children
21:                 if heapVals[left[r]] < heapVals[right[r]] then
22:                     r ←PRVROTATELEFT(r)
23:                     left[r] ←PRVREMOVE(left[r], ntbd, k)
24:                 else
25:                     r ←PRVROTATERIGHT(r)
26:                     right[r] ←PRVREMOVE(right[r], ntbd, k)
27:                 end if
28:             end if
29:         end if
30:     end if

31:     if r ≠ NULL_INDEX then
32:         PRVADJUST(r)
33:     end if

34:     return r
35: end procedure
```

What to turn in

Turn in your source code and **`Makefile`**. If you are using an IDE, compress the folder containing the project and submit that.