Data Structures and Objects
CSIS 3700

Fall Semester 2022 — CRN 41416

Project 4 — BST Dictionary
Due date: Monday, December 12, 2022

Goal

Develop and implement a dictionary that uses binary search trees 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 binary search tree. 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.

▹*Setting up the class*

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

```
#ifndef _BST_DICTIONARY_H
#define _BST_DICTIONARY_H

#include <cstdint>      // for uint32_t
#include <stdexcept>    // for domain_error

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

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

    }

    ~BSTDictionary() {

    }

    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:
    uint32_t prvAllocate() {

    }

    void prvFree(uint32_t n) {

    }

    void prvClear(uint32_t r) {

    }

    void prvAdjust(uint32_t r) {

    }

    uint32_t prvInsert(uint32_t r,uint32_t &n,const KeyType &k) {

    }

    uint32_t prvRemove(uint32_t r,uint32_t &ntbd,const KeyType &k) {

    }
};

#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.

- `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 nine 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:

```
    uint32_t
        root;           // tree root

    static uint32_t     // this is the shared data
        *counts,        // counts for each node
        *heights,       // heights for each node
        *left,          // left node indexes
        *right,         // right node indexes
        nTrees,         // number of BSTs with this key and value type
        capacity,       // size of the arrays
        freeListHead;   // the head of the unused node list (the free list)

    static KeyType
        *keys;          // pool of keys
    static ValueType
        *values;        // pool of values
```

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

```
template <typename KeyType,typename ValueType>
uint32_t *BSTDictionary<KeyType,ValueType>::counts = nullptr;
```

The other eight 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 BST dictionary

Preconditions None

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

```
1: procedure BSTDictionary($cap = INITIAL\_DEFAULT\_CAPACITY$)
2:     if $nTrees = 0$ then
3:         Allocate space for the six arrays          ▷ $cap$ is the initial size of all arrays
4:         $capacity \leftarrow cap$
5:         Generate the free list
6:     end if
7:     $nTrees \leftarrow nTrees + 1$
8:     $root \leftarrow NULL\_INDEX$
9: end procedure
```

Generating the free list

Note that this is not a standalone function; this is part of the constructor and `prvAllocate()` functions.

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 $i \leftarrow start$ to $end - 1$ do
3:         $left[i] \leftarrow i + 1$
4:     end for
5:     $left[end] \leftarrow NULL\_INDEX$
6:     $freeListHead \leftarrow start$
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

```
procedure ~BSTDictionary
    $nTrees \leftarrow nTrees - 1$

    if $nTrees = 0$ then
        Delete all shared arrays
    else
        prvClear($root$)
    end if
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

```
procedure Search($k$)
    $n \leftarrow root$
    while $n \neq NULL\_INDEX$ do
        if $k = keys[n]$ then
            return $values[n]$
        else if $k < keys[n]$ then
            $n \leftarrow left[n]$
        else
            $n \leftarrow right[n]$
        end if
    end while

    throw domain_error("Search: Key not found")
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:     $tmp \leftarrow$ PRVALLOCATE()

3:     $n \leftarrow tmp$

4:     $root \leftarrow$PRVINSERT($root, n, k$)

5:     if $n \neq tmp$ 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 $2 \cdot capacity$ 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:         $capacity \leftarrow 2 \cdot capacity$
9:     end if

10:    $tmp \leftarrow freeListHead$
11:    $freeListHead \leftarrow left[freeListHead]$

12:    $left[tmp] \leftarrow NULL\_INDEX$
13:    $right[tmp] \leftarrow NULL\_INDEX$
14:    $counts[tmp] \leftarrow 1$
15:    $heights[tmp] \leftarrow 1$

16:    return $tmp$
17: 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:     $left[n] \leftarrow freeListHead$

3:     $freeListHead \leftarrow n$
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:     $counts[r] \leftarrow$ GETCOUNT($left[r]$)+GETCOUNT($right[r]$) + 1
3:     $heights[r] \leftarrow \max($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. The main difference is that 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.

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 BST

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:         $keys[n] \leftarrow k$
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:         $left[r] \leftarrow$ PRVINSERT($left[r], n, k$)
10:    else
11:        $right[r] \leftarrow$ PRVINSERT($right[r], n, k$)
12:    end if
13:    PRVADJUST($r$)
14:    return $r$
15: end procedure
```

Recursively removing a node

Note that although this process is identical to recursive removal in a regular BST, it is presented here again since it's the most complicated action done in the course.

This is the recursive removal of a node from a regular BST, adapted to use simulated links instead of pointers.

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*

```
procedure PRVREMOVE($r$,$ntbd$,$k$)
    if $r = NULL\_INDEX$ then                         ▷ Subtree is empty, $k$ is not in tree
        throw domain_error("Remove: Key not found")
    end if

    $left[end] \leftarrow NULL\_INDEX$

    if $k < keys[r]$ then                             ▷ $k$ might be in left subtree
        $left[r] \leftarrow$ PRVREMOVE($left[r], ntbd, k$)
    else if $k > keys[r]$ then                        ▷ $k$ might be in right subtree
        $right[r] \leftarrow$ PRVREMOVE($right[r], ntbd, k$)
    else                                              ▷ $k$ is in node $r$
        $ntbd \leftarrow r$

        if $left[r] = NULL\_INDEX$ then
            if $right[r] = NULL\_INDEX$ then          ▷ $r$ is a leaf, subtree is removed
                $r \leftarrow NULL\_INDEX$
            else                                      ▷ $r$ only has right child, it is new root
                $r \leftarrow right[r]$
            end if
        else
            if $right[r] = NULL\_INDEX$ then          ▷ $r$ only has left child, it is new root
                $r \leftarrow left[r]$
            else                                      ▷ $r$ has two children
                Reduce two-child case to one-child case
            end if
        end if
    end if

    if $r \neq NULL\_INDEX$ then
        PRVADJUST($r$)
    end if

    return $r$
end procedure
```

Reducing two-child case

This algorithm takes the case where a node to be deleted has two children and converts it to a case where the node has at most one child, simplifying the removal process.

This is not a separate function; it is part of PRVREMOVE. Do not write it as a separate function.

Algorithm 11 Reducing the two-child case

Preconditions *r* and *ntbd* hold the index of a node to be deleted
r has two children

Postconditions *ntbd* holds the index of the node to be deleted
r is the root of the tree after removing *ntbd*

```
 1: if GETHEIGHT($right[r]$) > GETHEIGHT $left[r]$ then          ▷ Remove from the taller subtree
 2:     $tmp \leftarrow right[r]$                                ▷ Go to root of right subtree

 3:     while $left[tmp] \neq NULL\_INDEX$ do                    ▷ Move left as far as possible
 4:         $tmp \leftarrow left[tmp]$
 5:     end while

 6:     Swap $keys[r]$ and $keys[tmp]$
 7:     Swap $values[r]$ and $values[tmp]$

 8:     $right[r] \leftarrow$ PRVREMOVE($right[r], ntbd, k$)     ▷ Remove $k$ from right subtree
 9: else
10:     $tmp \leftarrow left[r]$                                 ▷ Go to root of left subtree

11:     while $right[tmp] \neq NULL\_INDEX$ do                   ▷ Move right as far as possible
12:         $tmp \leftarrow right[tmp]$
13:     end while

14:     Swap $keys[r]$ and $keys[tmp]$
15:     Swap $values[r]$ and $values[tmp]$

16:     $left[r] \leftarrow$ PRVREMOVE($left[r], ntbd, k$)       ▷ Remove $k$ from left subtree
17: end if
```

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.