University of Waterloo
CS240 Winter 2018
Assignment 4
Due Date: Wednesday, Mar. 14th, at 5pm

version: 2018-03-21 18:12

Please read the guidelines on submissions: `http://www.student.cs.uwaterloo.ca/~cs240/w18/guidelines.pdf`. This assignment contains written questions and a programming question. Submit your written solutions electronically as a PDF with the file named `a04wp.pdf` using MarkUs. We will also accept individual question files named `a04q1w.pdf`, `a04q2w.pdf`, …, `a04q5w.pdf` if you wish to submit questions as you complete them. For Question 5, submit your files `quadtreecompression.h` and `quadtreecompression.cpp` based on the skeleton files provided on the web page.

Problem 1 (2+2+6 marks)

(a) Give the contents of the hash table that results when you insert items with the keys

$$I, E, A, T, P, R, S, L, Y, G$$

in that order into an initially empty table of size $M = 16$. Use double hashing, with the hash functions given below:

c	A	E	G	I	L	P	R	S	T	Y
ASCII(c)	65	69	71	73	76	80	82	83	84	89
$h_1(c) = \text{ASCII}(c) \bmod 16$	1	5	7	9	12	0	2	3	4	9
$h_2(c) = (13 \cdot \text{ASCII}(c) \bmod 15) + 1$	6	13	9	5	14	6	2	15	13	3

For full credit it suffices to give the final correct answer, but we recommend showing some intermediate arrays to have a chance of part-marks in case of errors.

begin solution

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
A:										I							
						E				I							
		A				E				I							
		A			T	E				I							
	P	A			T	E				I							
	P	A	R		T	E				I							
	P	A	R	S	T	E				I							
	P	A	R	S	T	E				I			L				
	P	A	R	S	T	E				I			L				$h_1(Y) = 9$ occupied
	P	A	R	S	T	E				I			L				$h_1(Y) + h_2(Y) = 9 + 3 = 12$ occu
	P	A	R	S	T	E				I			L			Y	space at $h_1(Y) + 2h_2(Y)$
	P	A	R	S	T	E		G		I			L			Y	

————————end solution————————

(b) Find one character $c \in \{A, \ldots, Z\}$ that is different from all the ones in part (a) such that inserting c into your result from part (a) with the same hash-function would lead to "failure to insert" (i.e., to re-hashing). Briefly justify your answer.

Hint: Make use of the fact that the table-size M is not well-chosen for double-hashing.

————————begin solution————————

Consider adding character O. We have ASCII(0)=79, hence $h_1(O) = 15$ and $h_2(0) = 8$. So Insert(0) will try to put 0 at $A[15]$, but Y is already there. So it will try $h_1(0) + h_2(0) \bmod 16 = 15 + 8 \bmod 16 = 7$ next, but G is already there. And unfortunately the probe sequence is $(15, 7, 15, 7, \ldots)$ since M is not a prime number, so we never try any other spot and will (after n tries) return with failure, which means that we must re-hash the array.

————————end solution————————

(c) Suppose you have an implementation of double-hashing (not with the above hashing functions). Unfortunately there is a bug in your double-hashing code such that one or both of the hash functions always return the same integer[1] value $x > 0$. Describe what happens in each of these situations:

(i) h_1 has the bug,

(ii) h_2 has the bug,

(iii) both h_1 and h_2 have the bug.

You should comment on what the resulting probe sequence will be and how long you would expect that an Insert will take. (You need not prove the run-time of Insert formally, but compare its behaviour to hashing methods that we have seen in class.)

You may assume that M and the hashing-functions are suitable for double-hashing. In particular the load factor α is kept small ($< \frac{1}{2}$), M is a prime, $h_2(k) \in \{1, \ldots, M-1\}$, and h_1 and h_2 hash uniformly if they don't have the bug.

————————begin solution————————

(a) If h_1 has the bug, then key k gets the probe sequence

$$(x, x + h_2(k), x + 2h_2(k), \ldots)$$

[1]Blue color means a clarification to the assignment that was posted on piazza.

Thus initially all keys want the same slot (resulting in many collisions). However, since M is prime and $h_2(x) > 0$, this probe sequence will for any key explore all slots, and so insert will always be successful.

To see how long it will take, observe that we would expect $n/(M-1)$ keys to have the same hash-value $h_2(k)$, and hence the same probe-sequence. Since $\alpha < \frac{1}{2}$, we therefore expect a constant number of keys to have the same probe sequence. So this should not be worse that the other open addressing strategies, and in particular, have expected time $O(1)$ since α is kept small. It will be a bit slower than double-hashing, because we always one collision at the beginning.

(b) If h_2 has the bug, then key k will get the probe sequence

$$(h_1(k), h_1(k) + x, h_1(k) + 2x, \ldots)$$

If we are lucky and h_1 is a good hash function then we won't have many collisions in the first place. But even if we do, the probe sequence will explore all slots since $x > 0$ and M is a prime. This in fact acts a lot like linear probing (except that we advance by x slots rather than 1 slot). So this should again have expected time $O(1)$ for insert since α is kept small.

(c) If both h_1 and h_2 have the bug, then the probe sequence for *all* keys is

$$(x, 2x, 3x, 4x, \ldots)$$

Insert will still be successful because $x > 0$ and M is a prime and so the probe sequence explores all slots. But it will be very slow. Namely, the first inserted key will be at slot x, the second at slot $2x$, the ith one at slot ix. For the ith key to get to its slot, it has probed $i-1$ slots earlier. So to insert an item into a table with n elements will take $\Theta(n)$ time and be very slow.

———————end solution———————

Problem 2 (3+2+6 marks)

(a) Give the contents of the hash table that results when you insert items with the keys

$$I, E, A, T, P, R, S, L, Y, G$$

in that order into an initially empty table of size $M = 16$, using cuckoo hashing and the following hash functions (h_1 is the same as in the previous question):

c	A	E	G	I	L	P	R	S	T	Y
ASCII(c)	65	69	71	73	76	80	82	83	84	89
$h_1(c) = \text{ASCII}(c) \bmod 16$	1	5	7	9	12	0	2	3	4	9
$h_2(c) = 11 \cdot \text{ASCII}(c) \bmod 16$	11	7	13	3	4	0	6	1	12	3

For full credit it suffices to give the final correct answer, but we recommend showing some intermediate arrays to have a chance of part-marks in case of errors.

begin solution

The first few steps are the same until we have a collision since the same h_1 is used. Then:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
A:	P	A	R	S	T	E				I			L				
	P	A	R	S	T	E				Y			L				Y kicks out I
	P	A	R	I	T	E				Y			L				I kicks out S
	P	S	R	I	T	E				Y			L				S kicks out A
	P	S	R	I	T	E				Y		A	L				
	P	S	R	I	T	E		G		Y		A	L				

end solution

(b) Find one character $c \in \{A, \ldots, Z\}$ that is different from all the ones in part (a) such that inserting c into your result from part (a) with the same hash-function would lead to "failure to insert" (i.e., to re-hashing). (The character need not be the same one as in Q1(b).) Briefly justify your answer.

begin solution

Consider adding character C. We have ASCII(C)=67, hence $h_1(C) = 3$ and $h_2(C) = 1$. One could go through the steps of inserting to argue why this leads to failure, but an even easier argument is that the four character C, I, S, Y all have both their hash-function-values in $\{1, 3, 9\}$. Since no item ever goes outside the two possible slots, there is by pidgeon-hole principle no way in which we can insert C and find an empty spot for whoever gets kicked out.

end solution

(c) Suppose that you find that your hash table contains the following items:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	S		I	L	E	R	G		Y		A	T			P

You know that this hash table was obtained by inserting the keys with the above hash functions h_1 and h_2. You also know that no other key was every inserted, and no key was ever deleted. But you do not know in which order the insertions happened.

Give *three* distinct reasons (involving different keys) why this hash table could not possibly have been obtained via cuckoo hashing, no matter what the insertion order was.

begin solution

(a) P is at index 15. But $h_1(P) = h_2(P) = 0$, and in cuckoo hashing no key ever goes to a different slot than these two.

(b) R is at index 6$= h_2(R)$. It can go there only if someone else kicked it out of its preferred index $h_1(R) = 2$. But none of the other characters has hash-value 2.

(c) Consider keys L and T. L is at index 4$= h_2(L)$, so it must have been kicked out of its preferred index $h_1(L) = 12$ by someone. The only other key (among the given set) that hashes to 12 is is T. So T must have been inserted after L.

But now observe that T is at index 12$=h_2(T)$. As before, someone must have kicked T out of its preferred index $h_1(T) = 4$. The only other key (among the given set) that hashes to 4 is L. So L must have been inserted after T.

Not both the above statements are possible since we never delete and do not insert an element twice.

Optional observation: The remaining positions all could happen, e.g. consider insertion sequence I, E, A, S, Y, G, which results in

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A:										I						
						E				I						
		A				E				I						
		A		S		E				I						
		A		S		E				Y						Y kicks out I
		A		I		E				Y						I kicks out S
		S		I		E				Y						S kicks out A
		S		I		E				Y		A				
		S		I		E	G			Y		A				

end solution

Problem 3 (3+1+8 marks)

(a) Consider the set of 16 points in Figure 1.

Draw the corresponding kd-tree, following the conventions from the course slides, and starting by splitting along a suitable x-coordinate. You should show both the splitting lines that are inserted into the picture and the corresponding tree.

begin solution

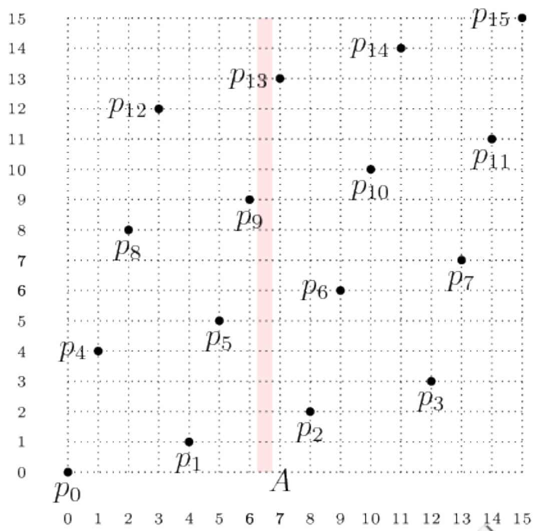
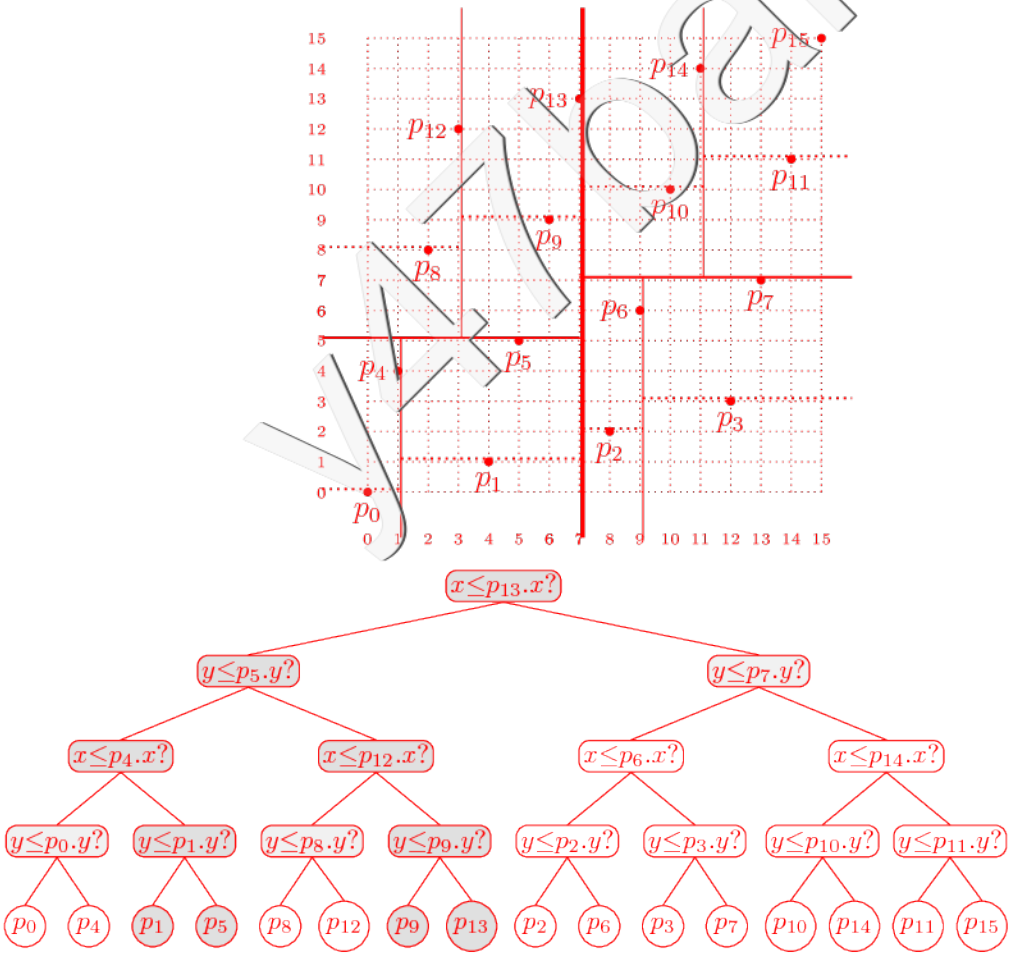


Figure 1: A set of points, and the query-rectangle A.



end solution

(b) Perform a range-query for the rectangle A indicated in Figure 1, which is rectangle $[7.25, 7.75] \times [0, 15]$ (following the pseudo-code on Module 8 slide 16).[2] You should show how the algorthm operates by indicating (in the tree that you created in part (a)) all nodes v such that `kdTree-RangeSearch`(v, A) was called on v. (We call such a node a "gray" node.)

begin solution

See the tree for part (a). Two shades of gray are used here; the darker gray are the "interesting" ones (in that the query-rectangle intersects their region but does not contain it), while the lighter gray ones are only gray due to the technicality that our pseudo-code recurses on the children before checking the region.

end solution

(c) Define a point set P for $n = 4^k$ (for some integer $k \geq 0$) as follows: $P = \{p_0, \ldots, p_{n-1}\}$, and for $\ell = 0, \ldots, n-1$

- p_ℓ has y-coordinate ℓ
- If $\ell = i\sqrt{n} + j$ (for some integers $i, j \geq 0$) , then p_ℓ has x-coordinate $j\sqrt{n} + i$.

(Figure 1 gives the point-set for $k = 2$.)

Show that there exists a query rectangle A such that no point of P lies in A, but there are $\Omega(\sqrt{n})$ gray nodes during the range-query for A.

(Motivation: Slide 18 of module 8 argued that the number of gray nodes is $O(\sqrt{n})$, so the purpose of this assignment is to argue that this bound is tight. You do *not* need to know the proof of the $O(\sqrt{n})$ bound to do this assignment.)

begin solution

Solution 1 (the more intuitive one, but only sketched here): The points form a "skewed grid", i.e., they look like a $\sqrt{n} \times \sqrt{n}$-grid except that rows/columns have been made slightly diagonal to have points in general position. The kd-tree of this will therefore (some details omitted) consist of rectangles that are also arranged as a $\sqrt{n} \times \sqrt{n}$-grid. Take as query-rectangle A that is a thickened vertical line with non-integer coordinate; clearly this contains no input-point. A will intersect a column of this grid of rectangles, hence $\sqrt{n}$ regions of leaves of the kd-tree. Each of these leaves

[2] The modules have not been finalized yet and slide-numbers may change by ± 1.

hence has a region R that intersects A but is not contained in it. All such leaves must be gray (because the recursive search only stops if $R \subseteq A$ or $R \cap A = \emptyset$). So there are $\sqrt{n}$ gray leaves, and hence $\Omega(\sqrt{n})$ gray nodes.

Solution 2 (less intuitive but cleaner): It actually turns out that the point-set is completely irrelevant; for *any* set of 4^k points in general position there exists a query-rectangle A such that $\Omega(\sqrt{n})$ nodes are gray. Namely, let A be any query-rectangle for which the y-range is so big that it contains *all* y-coordinates of points and the x-range is so small that it contains *none* of the y-coordinates of points. (In the example, $A = [2^{k-1} - \frac{3}{4}, 2^{k-1} - \frac{1}{4}] \times [-1, n]$ would do.) Clearly no point lies in A due to the x-coordinates.

We claim that a query for A will lead to at least $\sqrt{n}$ gray leaves (hence $\Omega(\sqrt{n})$ gray nodes. Clearly this holds if $k = 0$ (hence $n = 1$) because the root is always gray. Now consider $k > 0$ (hence $n \geq 4$) which means that the root has children and grandchildren. Of the two children, only one has a region that intersects A (because we split by x first); call this child c and let its children be g_1 and g_2. We recurse into c and from there into g_1 and g_2 because their regions are both intersected by A (because we split by y). Each grandchild g_1 and g_2 stores $n/4$ points, and A contains all y-coordinates and none of the x-coordinates of their points. By induction there are $\sqrt{n/4}$ gray leaves in g_1 and g_2 each, so in total the number of gray leaves is at least $2\sqrt{n/4} = \sqrt{n}$ as desired.

————————end solution————————

Problem 4 (7 marks)

We are given a set $\mathcal{W}$ of n of windows on the computer screen S (i.e., axis-parallel rectangles in the plane). A new window pops up and we wish to find all windows in $\mathcal{W}$ that are completely covered by the new window. Give an algorithm that finds all these windows in $O((\log n)^c + s)$ time, where s is the number of windows that are found, and $c \geq 1$ is a constant.

Formally, each window is described via a 4-tuple (x_ℓ, x_r, y_b, y_t) with $x_\ell < x_r$ and $y_b < b_t$, which corresponds to the rectangle $[x_\ell, x_r] \times [y_b, y_t]$. If the new window is $W' = (x'_\ell, x'_r, y'_b, y'_t)$, your query should return in $O((\log n)^c + s)$ time all those windows $W = (x_\ell, x_r, y_b, y_t)$ in $\mathcal{W}$ that satisfy $[x_\ell, x_r] \subseteq [x'_\ell, x'_r]$ and $[y_b, y_t] \subseteq [y'_b, y'_t]$.

To make this possible, you will need to assume that $\mathcal{W}$ is stored in a suitable data structure. Describe what you are using. This data structure should take at most $O(n \log^c n)$ space, and one should be able to build it in $O(n \log^c n)$ time.

————————begin solution————————

The crucial idea is that we can phrase this as a range-search problem. Namely, we have

$$[x_\ell, x_r] \subseteq [x'_\ell, x'_r] \text{ and } [y_b, y_t] \subseteq [y'_b, y'_t]$$

if and only if $x'_\ell \leq x_\ell$, $x_r \leq x'_r$, $y'_b \leq y_b$ and $y_t \leq y'_t$. This in turn holds if and only if

- $x_\ell \in [x'_\ell, \infty)$
- $x_r \in (-\infty, x'_r]$
- $y_b \in [y'_b, \infty)$, and
- $y_t \in (-\infty, y'_t]$

(It is also an option to use the "other" coordinate of W' in place of ∞, e.g. $x_\ell \in [x'_\ell, x'_r]$.)

Thus, view each window as a 4-dimensional point and then perform a range-query using a 4-dimenaional range-tree. The code is as follows:

```
CoveredWindowsPreprocessing(windows $\mathcal{W}$)
1.    build a 4-d range tree $T$ containing $\mathcal{W}$ (Slide 34)

CoveredWindows(4-d range tree $T$, window $W'$)
1.    // We assume that $T$ stores the points corresponding to $\mathcal{W}$
2.    let $W'$ correspond to the point $(x'_\ell, x'_r, y'_b, y'_t)$
3.    let $A$ be the 4-d rectangle $[x'_\ell, \infty) \times (-\infty, x'_r] \times [x'_b, \infty) \times (-\infty, y'_t]$
4.    return RangeQuery($T, A$) (Slide 34)
```

We know from class that the pre-processing time to build T is $O(n(\log n)^3)$ time, that T takes $O(n(\log n)^3)$ space, and that we can perform a query in $O((\log n)^4 + s)$ time. By the above equivalence, the query for A exactly returns all the covered windows in the required time with $c = 4$.

(Optional: Below is a more detailed description of the range tree and how to search in it.)

- The root-tree T_ℓ is a balanced BST-tree by the first coordinate (the left side of the rectangle).
- Each node v of T_ℓ has an associated tree $T_r(v)$ that is a balanced BST-tree of the descendants of v with respect to the second coordinate.
- Each node w of $T_r(v)$ has an associated tree $T_b(v, w)$ that is a balanced BST-tree of the descendants of w with respect to the third coordinate.
- Each node u of $T_b(v, w)$ has an associated tree $T_t(v, w, u)$ that is a balanced BST-tree of the descendants of u with respect to the fourth coordinate.

Searching: Given the new window $W' = (x'_\ell, x'_r, y'_b, y'_t)$, we need to do a range-search in this range tree. Proceed as follows:

- Search for range $[x'_\ell, \infty)$ in T_ℓ and find all boundary and allocation nodes. Any boundary node can be tested for being in the range directly—there are only $O(\log n)$ of them. For any allocation node (there are $O(\log n)$ of them), proceed with the next step.

- For any allocation node v in T_ℓ, search for range $(-\infty, x'_r]$ in $T_r(v)$ and find all boundary and allocation nodes. Any boundary node can be tested for being in the range directly—there are only $O(\log n)$ of them. For any allocation node (there are $O(\log n)$ of them), proceed with the next step.

- For any allocation node w in $T_r(v)$, search for range $[y'_b, \infty)$ in $T_b(v,w)$. As before boundary nodes can be tested directly, and for any allocation node (there are $O(\log n)$ of them), proceed with the next step.

- For any allocation node u in $T_b(v,w)$, search for range $(-\infty, y'_t]$ in $T_t(v,w,u)$. As before any boundary nodes can be tested directly. For any allocation node, we claim that *all* descendants are windows that are covered by W'. Namely, let (x_ℓ, x_r, y_b, y_t) be such a descendant, say at node z. We know $x'_\ell \leq x_\ell$, because v was an allocation node in T_ℓ and hence z (which is a descendant of v) lies within the range of the search $[x'_\ell, \infty)$ in that tree. Similarly one argues that $x_r \leq x'_r$ and $y'_b \leq y_b$ and $y_t \leq y'_t$.

  So report all descendants of all allocation nodes. This takes $O(s)$ time overall, since every window is reported only once.

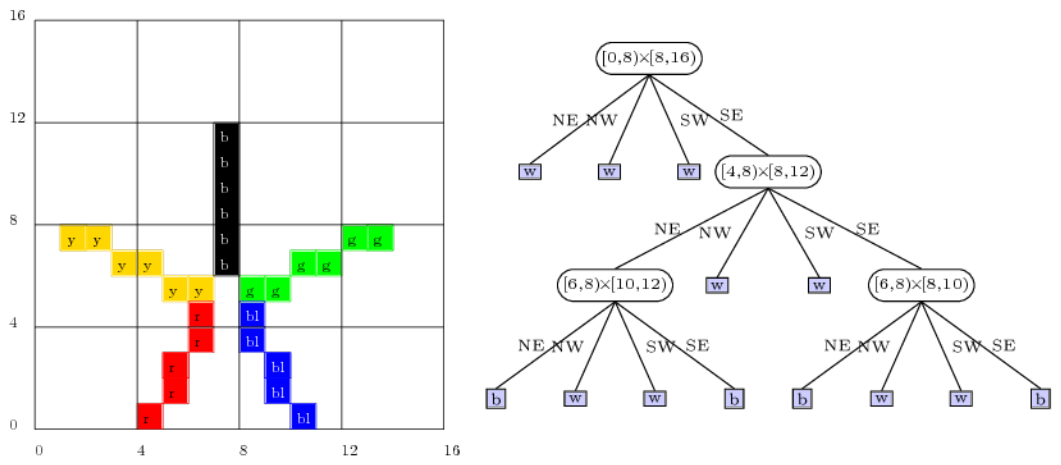
Overall, we need at most $O(\log n^3)$ range queries, each of which take $O(\log n)$ time, leading to a run-time of $O(\log^4 n + s)$.

——————————end solution——————————

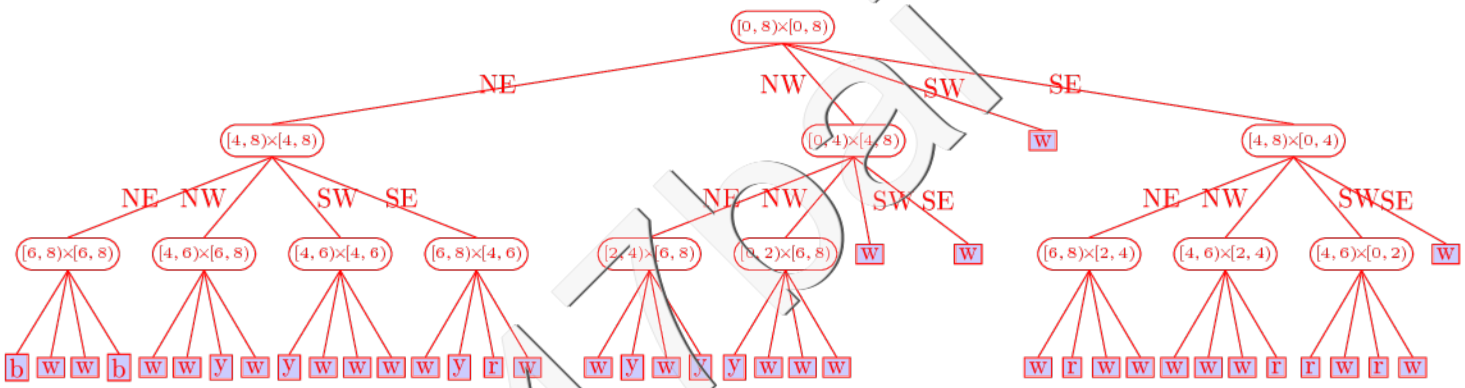
Problem 5 (2+24(+10) marks)

One of the applications of quadtrees is compression of pictures. The picture is recursively divided in quadrants until the entire quadrant is of the same colour.

(a) (Written) Using this rule, give the quad-tree that represents the region $[0,8) \times [0,8)$ of the figure given below. To explain the drawing conventions, we give you below the quad-tree for the region $[0,8) \times [8,16)$ of the figure. (You may omit labels NE, NW, etc., especially near leaves.) Any unlabelled pixel is white ('w').



———————begin solution———————



———————end solution———————

(b) (Programming)

Write a program that converts a given (black-and-white) picture into a quadtree that represents it as described in part (a), and vice versa. (Details are below.)

———————begin solution———————

A sample solution will be posted on the web page.

———————end solution———————

(c) (Bonus)

Comment on the effectiveness of this method of compressing pictures. To support your arguments, provide experimental results (using your own implementation) that compare $|A|$ and $|T|$ for various pictures. Coming up with methods of generating pictures and what values of n to test are part of the question.

——————begin solution——————

We do not provide an actual solution here, only some ideas of what to expect. Nearly everything here depends on the choice of picture. If you test on randomly generated 01-matrices, then the compression should be quite bad; sometimes the quad-tree might even be bigger than the size of the array because of its interior nodes. On real-life pictures the compression should be quite good, since they typically have lots of solid white or solid black.

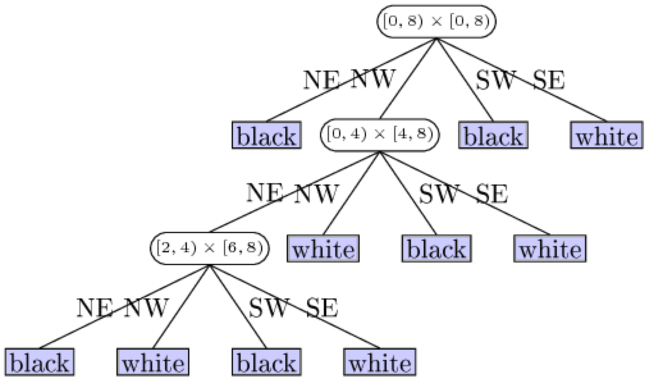
——————end solution——————

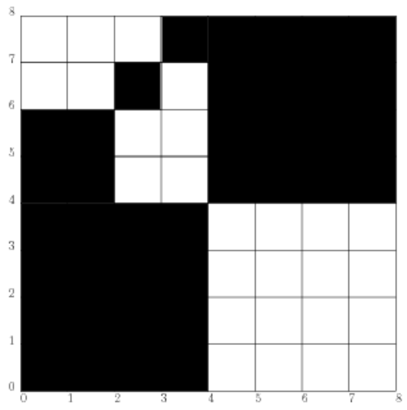
Details: On the web page, you will find a stub-file `quadtreecompression.h` and `quadtreecompression.cc`.

- Implement `string * arrayToTree(int n, int * A)`. Here n is a power of 2, array A has size n^2 and each entry is 0 or 1. The function returns a reference to a string of a quad-tree that represents the same picture.
- Implement `int * treeToArray(int n, string * T)`. Here n is a power of 2, T is the string of a quad-tree that stores a picture of size $n \times n$, where all leaves are 'b' or 'w'. The function returns an array that represents the same picture.
- A 01-array $A[0..n^2-1]$ stores a black-and-white picture as follows: For $0 \leq x, y < n$, entry $A[\texttt{indexOf}(n,x,y)]$ stores the color of the pixel whose bottom-left corner is (x,y), Here `indexOf` is a function provided in the stub. We use the convention that 0 represents white and 1 represents black.
- A string T represents a quadtree by having a character for each node in pre-order. A character 'i' means an internal node, a character 'b' means a leaf that is black and a character 'w' means a leaf that is white. String T lists the nodes in pre-order, i.e., it first has the character of the root, then recursively the characters of the NE-child, the characters of the NW-child, the characters of the SW-child, and the characters of the SE-child. For example, if $n = 8$, then

$$T = (i, b, i, i, b, w, b, w, w, b, w, b, w)$$

represents the following quad-tree (and picture):





- With the above conventions and for fixed n, one can argue that an array A determines a unique string T and vice versa (you need not prove this). Figuring out how to do the conversion between them is part of your assignment.
- The stubs contain "`#ifndef SCRIPTS`" and "`#endif`" at various places. Do not remove these. Anything that is surrounded by "`#ifndef SCRIPTS`" and "`#endif`" will not be translated by our testing scripts, and you can add testing-routines within them.
- The stub-file contains some printing routines that you may use, but do not have to use. You can add other functions as you see fit.
- You should not use (or need) any libraries beyond the ones provided in the stub-files.

```
#include <iostream>
#include <string>
using namespace std;

class quadTreeCompression {
private:
    // You can add any private methods here
    int printTree(string * A, int indent, int parseIndex);
    void printArray(int n, int * A, int l, int r, int b, int t);
    int indexOf(int n, int x, int y);

    void fill (int n, int * A, int l, int r, int b, int t, int v);
    int dominant (int n, int * A, int l, int r, int b, int t);

    void subtreeToArray(int n, string * T, int &index, int l, int r, int b, int t
        , int * A);
    void subarrayToTree(int n, int * A, int l, int r, int b, int t, string * T);

public:
    // Do not change the public interface
    quadTreeCompression();
    ~quadTreeCompression();

    string * arrayToTree(int n, int * A);
    int * treeToArray(int n, string * T);

    // print functions
    void print(int n, int * A);
    void printArray(int n, int * A);
    void printTree(string * T);
};
```

```
#include "quadtreecompression.h"

quadTreeCompression::quadTreeCompression() {

}

quadTreeCompression::~quadTreeCompression() {

}

int quadTreeCompression::indexOf(int n, int x, int y){
// pre: 0 <= x,y < n
// post: return the 1D index representation of (x, y) in an nxn-array

    return n * y + x;
}

void quadTreeCompression::fill (int n, int * A, int l, int r, int b, int t, int v
    ){
// pre: n is a power of 2, n>=1, A is an nxn-array of 0 or 1
//      0 <= l,r,b,t < n, v is 0 or 1
// post: fills each (x,y) in A with v (where l<=x<r, b<=y<t)

    for(int j=b; j<t; j++){
        for(int i=l; i<r; i++){
            A[indexOf(n, i, j)] = v;
        }
    }
}

int quadTreeCompression::dominant (int n, int *A, int l, int r, int b, int t){
// pre: n is a power of 2, n>=1, A is an nxn-array of 0 or 1
//      0 <= l,r,b,t < n
// post: if each (x,y) in A has the same value, return that value
//       otherwise return -1 (where l<=x<r, b<=y<t)

    int elem = A[indexOf(n, l, b)];

    for(int j=b; j<t; j++){
        for(int i=l; i<r; i++){
            if(elem != A[indexOf(n, i, j)]){
                elem = -1; //no dominant element
            }
        }
    }
    return elem;
}

int* quadTreeCompression::treeToArray(int n, string * T) {
// pre: n is a power of 2, n>=1, T is a string that stores
//      a picture in the form of a quadtree
// post: return an nxn-array of 0 or 1 representing T's picture

    int * A = new int[n*n];

    int index = 0;
    subtreeToArray(n, T, index, 0, n, 0, n, A);

    return A;
}

void quadTreeCompression::subtreeToArray(int n, string * T, int &index, int l,
    int r, int b, int t, int * A){
// pre: n, is a power of 2, n>=1,
//      T is a string that stores a picture in the form of a quadtree,
//      A is an nxn-array of 0 or 1,
//      0 <= l,r,b,t < n, (r-l) = (t-b) are powers of 2 and >=1,
//      0 <= index < T->length
// post: modifies A in the range (l..r-1, b..t-1) to represent
//       the node at the index'th position of T

    char cur = T->at(index);
    index++;

    if(cur == 'w'){
        fill(n, A, l, r, b, t, 0);
    }else if(cur == 'b'){
        fill(n, A, l, r, b, t, 1);
    }else{
        int midx = (r + l)/2;
        int midy = (b + t)/2;
        subtreeToArray(n, T, index, midx, r, midy, t, A);
        subtreeToArray(n, T, index, l, midx, midy, t, A);
        subtreeToArray(n, T, index, l, midx, b, midy, A);
        subtreeToArray(n, T, index, midx, r, b, midy, A);
    }
}

string * quadTreeCompression::arrayToTree(int n, int * A) {
// pre: n is a power of 2, n>=1, A is an nxn-array containing 0 or 1
// post: return string that stores picture in A in form of quadtree

    string * T = new string;
    *T = "";

    subarrayToTree(n, A, 0, n, 0, n, T);

    return T;
}

void quadTreeCompression::subarrayToTree(int n, int * A, int l, int r, int b, int
    t, string * T){
// pre: n is a power of 2, n>=1.
//      T is a string that stores a picture in the form of a quadtree,
//      A is an nxn-array of 0 or 1,
//      0 <= l,r,b,t < n, (r-l) = (t-b) are powers of 2 and >=1,
// post: modifies T to store the picture of A in the range (l..r-1, b..t-1)

    int dom = dominant(n, A, l, r, b, t);

    if(dom == 0){
            T->append("w");
        }else if(dom == 1){
            T->append("b");
        }else{
            T->append("i");
            int midx = (r + l)/2;
            int midy = (b + t)/2;

            subarrayToTree(n, A, midx, r, midy, t, T);
            subarrayToTree(n, A, l, midx, midy, t, T);
            subarrayToTree(n, A, l, midx, b, midy, T);
            subarrayToTree(n, A, midx, r, b, midy, T);
        }
    }

    // The following are some printing functions to help you test

    void quadTreeCompression::printArray(int n, int * A) {
        printArray(n,A,0,n,0,n);
    }

    void quadTreeCompression::printArray(int n, int * A, int l, int r, int b, int t)
        {
    // pre: A is an nxn-array
        int i,j;
        cout << "/"; for (i=0; i<n; i++) cout << "-"; cout << "\\\n";
        for (j=n-1; j>=0; j--) {
            cout << "|";
            for (i=0; i<n; i++) {
                if (l<=i && i<r && b<=j && j<t) {
                    if (A[indexOf(n, i, j)]==1)
                        cout << "*";
                    else if (A[indexOf(n, i, j)]==0)
                        cout << "0";
                    else
                        cout << "?";
                }
                else cout << " ";
            }
            cout << "|\n";
        }
        cout << "\\"; for (i=0; i<n; i++) cout << "-"; cout << "/\n";
    }

    void quadTreeCompression::printTree(string * T) {
        printTree(T,0,0);
    }

    int quadTreeCompression::printTree(string * T, int parseIndex, int indent) {
    // prints tree starting from T[parseIndex] with indent as passed
    // returns last index that was parsed
        for (int i = 0; i < indent; i++) cout << " ";
        cout << T->at(parseIndex) << "\n";
        if (T->at(parseIndex) == 'i') {
            parseIndex = printTree(T,parseIndex+1,indent+5);
            parseIndex = printTree(T,parseIndex+1,indent+5);
            parseIndex = printTree(T,parseIndex+1,indent+5);
            parseIndex = printTree(T,parseIndex+1,indent+5);
        }
        return parseIndex;
    }

    // Anything within the following directive will not be
    // run by our testing scripts.
    #ifndef SCRIPTS
    int main() {
        // You can include any testing routines here.
        // You can find a sample testing methodology below.

        quadTreeCompression * h = new quadTreeCompression();

        string * T = new string("ibiibwbwwbwbw");

        cout << "The original tree is:\n";
        h->printTree(T);

        int n = 8;
        int * A = h->treeToArray(n,T);

        cout << "\n\nConverting this tree into an array gives:\n";
        h->printArray(n,A);

        T = h->arrayToTree(n, A);

        cout << "\n\nConverting this array back to a tree gives:\n";
        h->printTree(T);

        delete h;
    }
    #endif
```