PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)
100-ft Ring Road, Bengaluru – 560 085, Karnataka, India
DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGG

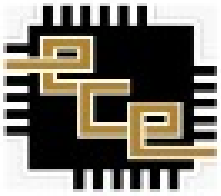
PROGRAM B.TECH

RISC-V Architecture (UE20EC302)

Project Report on

Implementation of Breadth First Search Algorithm

On C/ASM.



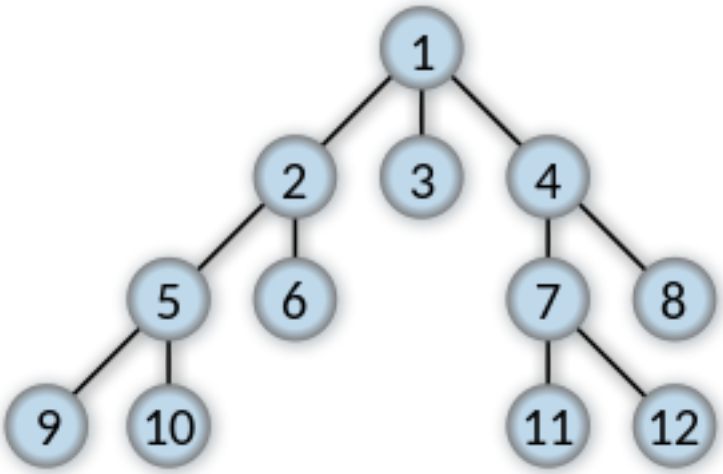
Submitted By

– B Gautham PES1UG20EC044

– Dheemanth R Joshi PES1UG20EC059

Problem Statement: Implement Breadth First search algorithm on the risc-v processor to analyse the performance.

Introduction:
Breadth-first search (**BFS**) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a *queue* is needed to keep track of the child nodes that were encountered but not yet explored.



NOTE: Our implementation constraints each node to have atmost two child nodes. A graph following this constraint is often called a *"Binary Tree".*

Breadth-first search can be generalized to graphs, when the start node (sometimes referred to as a 'search key') is explicitly given, and precautions are taken against following a vertex twice.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

Traversing child nodes

A graph can contain cycles, which may bring you to the same node again while traversing the graph. To avoid processing of same node again, use a boolean array which marks the node after it is processed. While visiting the nodes in the layer of a graph, store them in a manner such that you can traverse the corresponding child nodes in a similar order.

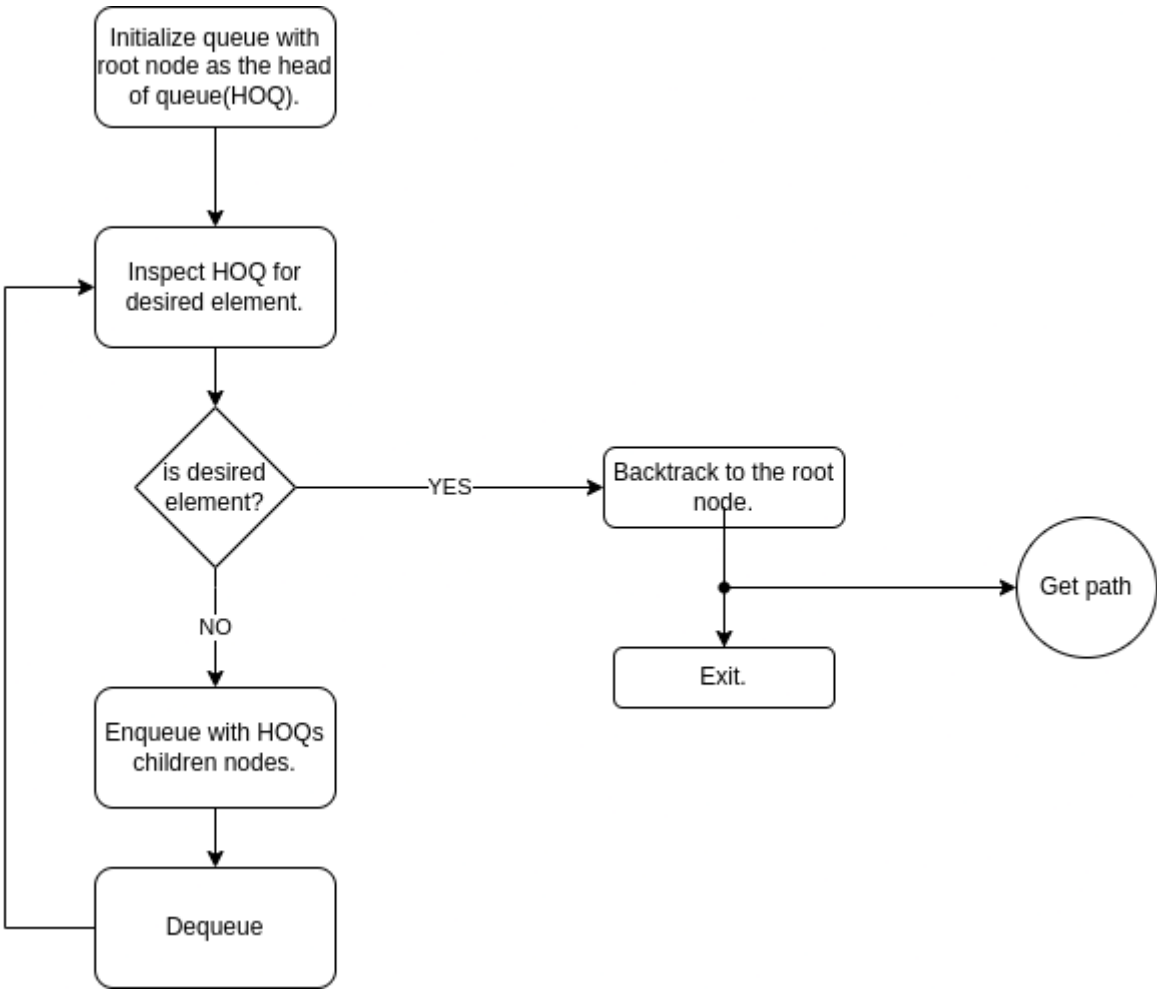
To make this process easy, use a queue to store the node and mark it as 'visited' until all its neighbours (vertices that are directly connected to it) are marked. The queue is a data-structure that follows the First In First Out (FIFO) queuing method, and therefore, the

neighbours of the node will be visited in the order in which they were inserted in the node i.e. the node that was inserted first will be visited first, and so on.

Methodology => Implementation of BFS algorithm:

We made use of pointers to link a parent node to its children. Queues can be implemented by the use of a *singly linked list.*
Brief overview of the working of BFS algorithm is written below:
* Declare a queue and insert the starting vertex.
* Initialize a visited array and mark the starting vertex as visited.
* Follow the below process till the queue becomes empty:
* Remove the first vertex of the queue.
* Mark that vertex as visited.
* Insert all the unvisited neighbours of the vertex into the queue.

Flow chart:
(Head of queue(HOQ) : Element at the start of the queue.
Enqueue : Add elements to the rear of the queue.
Dequeue: Remove the element at the start of the queue(HOQ). Now the second element in the queue becomes the HOQ.
)



Code and Implementation:

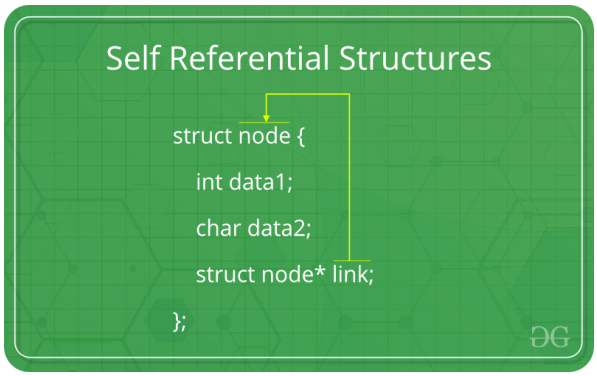
(All files are available @ https://github.com/bgautham4/RISC
Assembly code is below Implementation details).

```
typedef struct Node{
struct Node *p; //Parent node
char elem; //element.
struct Node *c1; //Child node 1
struct Node *c2; //Child node 2
}Node;


typedef struct Node_queue{  //Definition of the Queue using linked list.
struct Node node;
struct Node_queue *next;
}Nqueue;


void enqueue(Nqueue *head,Node node){
Nqueue *next = head->next;
Nqueue *new = (Nqueue *)malloc(sizeof(Nqueue)); //Allocate memory for the new element to be added to
rear of queue.
if (new==NULL){
printf("An error in mem allocation occured! Exiting..");
return;
}
else{
new->node = node;
new->next = NULL;
}
while (next!=NULL){//Traverse to the end of queue.
head = next;
next = head->next;
}
head->next = new; //Link last ement of queue with new element.
return;
}


void dequeue(Nqueue **head_ptr){
if(*(head_ptr)==NULL){
printf("Queue is already empty!");
return;
}
else{
Nqueue *temp = (*head_ptr)->next;
Nqueue *del = (*head_ptr); // del points to the HOQ.
*head_ptr = temp; //The above lines of code reassign the head of the queue to be the next element in
queue.
free(del); //Deallocate memory for the removed element to prevent memory leaks.
return;
```

```
}
}

char * BFS(Node *root,char elem){
Nqueue *queue = (Nqueue *)malloc(sizeof(Nqueue)); //Allocate memory for the queue.
char *path = (char *)malloc((int) sizeof(char)*10); //Create an empty array of characters (string) of length
10. This will hold the path to the node to be searched.
Node *child1 = root->c1;
Node *child2 = root->c2;
Node *backtrack_node;
char check = root->elem;
queue->node = *(root);
queue->next = NULL;
while (check != elem){
if (child1 != NULL){
enqueue(queue,*(child1));
}
if (child2 != NULL){
enqueue(queue,*(child2));
}
dequeue(&queue);
check = queue->node.elem;
child1 = queue->node.c1;
child2 = queue->node.c2;
}
*(path) = queue->node.elem;
int i = 1;
backtrack_node = queue->node.p;
while (backtrack_node != NULL){
*(path + i) = backtrack_node->elem;
backtrack_node = backtrack_node->p;
i++;
}
*(path+i) = '\0';
while (queue != NULL){
dequeue(&queue); //Empty queue and free up memory.
}
return path;
}
```

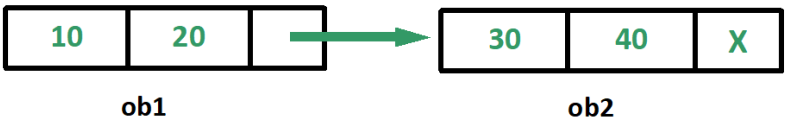
Implementation details:

Self referential structures:

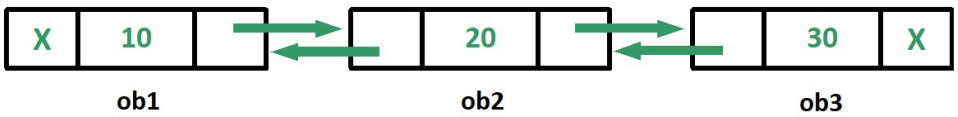


Self referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.

The program relies heavily on the use of these self refrential structures, which can be seen in the code for the definition of the nodes(multiple links) and queues which use singly linked lists.



Single link



Multiple
links

Implementation of Queues in RISC-V ASM:

```
main:
li s0,0x2000 #use saved register s0 to store the start of queue.
li t0,23
sw t0,0(s0) #Start queue with some random element, say 23
li t1,0x0 #Define NULL to equal 0.
sw t1,4(s0) #Set the queue's end by declaring the last reference to be
NULL.

jal x1,traverse #View the contents of the queue (reg t0 contains the
elements).
#Let us now try to enqueue an element into the queue
li a0,12 #Load a value into the argument register and call enqueue sub-
routine
jal x1,enqueue
jal x1,traverse
#Let us enqueue some more elements.
```

```
Addelems:
	li a0,10
	jal x1,enqueue
	li a0,14
	jal x1,enqueue
	li a0,17
	jal x1,enqueue
jal x1,traverse
#Let us now try to dequeue elements from the queue.
Rmvelems:
	jal x1,dequeue
	jal x1,dequeue
	jal x1,dequeue
jal x1,traverse
j Exit

enqueue:
	mv t1,s0 #Start of queue
	lw t2,4(s0) #next element of queue
	beq t2,x0,Append
	Loop1:
		mv t1,t2
		lw t2,4(t2)
		bne t2,x0,Loop1 #Keep looping until t2 hits NULL.
	Append:
		addi t3,t1,8 #Mem location for the new element.
		sw t3,4(t1) #Make the previous last element point to this
newly created elem
ent.
		sw a0,0(t3)
		sw x0,4(t3) #Make new element point to NULL.
		jalr x1 #Jump back to caller after adding new element.

dequeue:
	mv t1,s0 #Head of the queue
	lw t2,4(s0) #next element of the queue
	beq t1,x0,Exit_deq #If the queue is already empty, then exit.
	sw x0,0(t1) #Free up the element being removed
	sw x0,4(t1)
	mv s0,t2 #Head of the queue now points to the next element.
	Exit_deq:jalr x1
```

```
traverse:
        mv t1,s0
        Loop:
                lw t0,0(t1) #t0 contains the element.
                lw t1,4(t1)
                bne t1,x0,Loop #Loop until NULL is reached
                jalr x1 #Jump back to caller after traversal

Exit:nop
```

Outputs:

Note that all memory allocations have been made continous for better visualization. This need not be the case with queues as they are singly linked lists and do not require that references to each element be continous.

x8	s0	0x00002000

The location of the head of the queue held in register s0.

0x00002004	0x00000000	0x00	0x00	0x00	0x00
0x00002000	0x00000017	0x17	0x00	0x00	0x00

The queue in memory
0x2000 holds the element and 0x2004 holds the reference to the next element in the queue (in this case it is null).

```
#Let us now try to enqueue an element into the queue
li a0,12 #Load a value into the argument register and call enqueue sub-routine
jal x1,enqueue
jal x1,traverse
```

0x0000200c	0x00000000	0x00	0x00	0x00	0x00
0x00002008	0x0000000c	0x0c	0x00	0x00	0x00
0x00002004	0x00002008	0x08	0x20	0x00	0x00
0x00002000	0x00000017	0x17	0x00	0x00	0x00

After enqueuing an element.
Note how the reference of the first element @ 0x2004 has now been changed to 0x2008 which is the new element.

```
#Let us enqueue some more elements.
Addelems:
    li a0,10
    jal x1,enqueue
    li a0,14
    jal x1,enqueue
    li a0,17
    jal x1,enqueue
jal x1,traverse
```

0x00002024	0x00000000	0x00	0x00	0x00	0x00
0x00002020	0x00000011	0x11	0x00	0x00	0x00
0x0000201c	0x00002020	0x20	0x20	0x00	0x00
0x00002018	0x0000000e	0x0e	0x00	0x00	0x00
0x00002014	0x00002018	0x18	0x20	0x00	0x00
0x00002010	0x0000000a	0x0a	0x00	0x00	0x00
0x0000200c	0x00002010	0x10	0x20	0x00	0x00
0x00002008	0x0000000c	0x0c	0x00	0x00	0x00
0x00002004	0x00002008	0x08	0x20	0x00	0x00
0x00002000	0x00000017	0x17	0x00	0x00	0x00

Adding more elemets(enqueue) into
the queue.

```
#Let us now try to dequeue elements from the queue.
Rmvelems:
    jal x1,dequeue
```

x8	s0	0x00002008

0x00002024	0x00000000	0x00	0x00	0x00	0x00
0x00002020	0x00000011	0x11	0x00	0x00	0x00
0x0000201c	0x00002020	0x20	0x20	0x00	0x00
0x00002018	0x0000000e	0x0e	0x00	0x00	0x00
0x00002014	0x00002018	0x18	0x20	0x00	0x00
0x00002010	0x0000000a	0x0a	0x00	0x00	0x00
0x0000200c	0x00002010	0x10	0x20	0x00	0x00
0x00002008	0x0000000c	0x0c	0x00	0x00	0x00
0x00002004	0x00000000	0x00	0x00	0x00	0x00
0x00002000	0x00000000	0x00	0x00	0x00	0x00

Element has been removed(dequeued)
Note the contents of 0x2000 and 0x2004
which is equal to 0 and the contents of reg s0

which is now 0x2008.

```
    jal x1,dequeue
    jal x1,dequeue
jal x1,traverse
j Exit
```

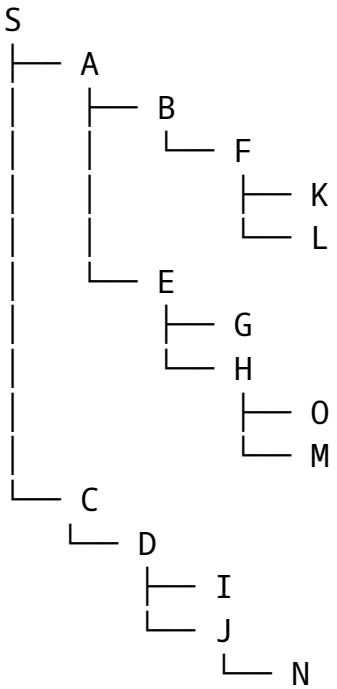
x8	s0	0x00002018

0x00002024	0x00000000	0x00	0x00	0x00	0x00
0x00002020	0x00000011	0x11	0x00	0x00	0x00
0x0000201c	0x00002020	0x20	0x20	0x00	0x00
0x00002018	0x0000000e	0x0e	0x00	0x00	0x00
0x00002014	0x00000000	0x00	0x00	0x00	0x00
0x00002010	0x00000000	0x00	0x00	0x00	0x00
0x0000200c	0x00000000	0x00	0x00	0x00	0x00
0x00002008	0x00000000	0x00	0x00	0x00	0x00
0x00002004	0x00000000	0x00	0x00	0x00	0x00
0x00002000	0x00000000	0x00	0x00	0x00	0x00

After dequeuing more elements
from the queue.

BFS:

This is the defined binary tree.



```
Enter the node to find (Note: Input is case sensitive!!) (A-O) : H
Path to node H is (S is the root node):S A E H

Enter the node to find (Note: Input is case sensitive!!) (A-O) : N
Path to node N is (S is the root node):S C D J N

Enter the node to find (Note: Input is case sensitive!!) (A-O) : B
Path to node B is (S is the root node):S A B
```

References:

Optimizations : https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Search algorithms : https://www.youtube.com/watch?v=j1H3jAAGlEA&t=336s

Self referential structures : https://www.geeksforgeeks.org/self-referential-structures/