ALGORITHM AND ANALYSIS (COSC1285)

ASSIGNMENT 1

1. THEORETICAL ANALYSIS

Annotation: rq = The ordered array runqueue
Bst = Binary Search Tree

Array runqueue				
Scenarios	Method Structure	Best Case	Worse Case	Big O
Growing Runqueue (enqueue)	Find whether the new process is already in the rq. A for loop (1 to n). (FYI:Process not in the rq scenario is assumed here)	$C_b(n) = n$ Ex: Process is not in rq	$C_w(n) = n$ Ex: Process is not in rq	$O(n)$
	A for loop (1 to n) to find the index, where the new process should be inserted.	$C_b(n) = 1$ Ex: vt <= minimum vt in rq	$C_w(n) = n$ Ex: vt >= maximum vt in rq	$O(n)$
	Creating the new rq with new process added- A for loop (1 to n+1)	$C_b(n) = 2n+2$	$C_w(n) = n+1$	$O(n)$
	Total	$C_b(n) = n+3$	$C_w(n) = 3n+1$	$O(n)$
Shrinking Runqueue (dequeue)	Delete the 1st element in the array. Creating the new rq with 1st element deleted- A for loop (1 to n-1)	$C_b(n) = n-1$	$C_w(n) = n-1$	$O(n)$
	Total	$C_b(n) = n-1$	$C_w(n) = n-1$	$O(n)$
Calculate total vruntime of preceding processes in Runqueue	Find the location of the process. A for loop (1 to n)	$C_b(n) = 1$ Ex: Process is in the 1st position	$C_w(n) = n$ Ex: Process is in the last position	$O(n)$
	Calculate the pt: A for loop (1 to n)	$C_b(n) = 1$ Ex: Process is in the 1st position	$C_w(n) = n$ Ex: Process is in the last position	$O(n)$
	Total	$C_b(n) = 2$	$C_w(n) = 2n$	$O(n)$

LinkedList runqueue (Doubly LinkedList)				
Scenarios	Method Structure	Best Case	Worse Case	Big O
Growing Runqueue (enqueue)	Find whether the new process is already in the rq. A while loop (1 to n). (FYI:Process not in the list scenario is assumed here)	$C_b(n) = n$ Ex: Process is not in the list	$C_w(n) = n$ Ex: Process is not in the list	$O(n)$
	Find the index where the new process should be added. A for loop (1 to n).	$C_b(n) = 1$ Ex: vt <= head.vt	$C_w(n) = n$ Ex: vt >= maximum vt in list	$O(n)$
	Insert the new process. A for loop (1 to n/2).	$C_b(n) = 1$ Ex: vt<=head.vt or vt>=tail.vt	$C_w(n) = n/2$ Ex: vt is the median	$O(n)$
	Total	$C_b(n) = n+2$	$C_w(n) = 2.5n$	$O(n)$
Shrinking Runqueue (dequeue)	Delete the head node.	$C_b(n) = 1$	$C_w(n) = 1$	$O(1)$
	Total	$C_b(n) = 1$	$C_w(n) = 1$	$O(1)$
Calculate total vruntime of preceding processes in Runqueue	Calculate the pt: A while loop (1 to n)	$C_b(n) = 1$ Ex: Process is the head node	$C_w(n) = n$ Ex: Process is in the tail node	$O(n)$
	Total	$C_b(n) = 1$	$C_w(n) = n$	$O(n)$

Binary Search Tree runqueue				
Scenarios	Method Structure	Best Case	Worse Case	Big O
Growing Runqueue (enqueue)	Find whether the new process is already in the bst. In order traversal (recursive method)	$C_b(n) = n$ Ex: Process is not in the bst	$C_w(n) = n$ Ex: Process is not in the bst	$O(n)$
	Insert process to the runqueue. h is the height of the tree	$C_b(n) = 1$ Ex: Adding the process as root	$C_w(n) = n$ Ex: Skewed tree, deepest leaf node.	$O(n)$
	Total	$C_b(n) = n+1$	$C_w(n) = 2n$	$O(n)$

Shrinking Runqueue (dequeue)	Traverse to the leftmost node. Case 1: Node is a leaf (parent.leftChild = null) Case 2: Node has a right subtree (parent.leftChild = node.rightChild)	$C_b(n) = 1$ Ex: Right skewed tree(i.e. Root is the minimum element)	$C_w(n) = n$ Ex: Left skewed tree, deepest leaf node. Height = n	$O(n)$
	Total	$C_b(n) = 1$	$C_w(n) = n$	$O(n)$
Calculate total vruntime of preceding processes in Runqueue	Find the process of the given procLabel. In order traversal (recursive method)	$C_b(n) = 1$ Ex: Process is in the leftmost node	$C_w(n) = n$ Ex: Process is in the rightmost node	$O(n)$
	Calculate pt. In order traversal (recursive method)	$C_b(n) = 1$ Ex: Process is in the leftmost node	$C_w(n) = n$ Ex: Process is in the rightmost node	$O(n)$
	Total	$C_b(n) = 2$	$C_w(n) = 2n$	$O(n)$

- According to the above tables, the most efficient data structure for Growing Runqueue scenario is the Binary search tree, and for Shrinking Runqueue the Ordered Linked list wins with constant time. In the 3rd scenario of Calculating total vruntime of preceding processes again Ordered Linked list gains the lowest time complexity.
- In the array data structure, when enqueuing and dequeuing, a new array needs to be generated by copying elements, and this costs a for loop running n times.
- Doubly linked list earned better time efficiency when enqueuing new processes. However, it was not able to win over the Binary search tree, enqueue method. Furthermore, in scenario 2 (dequeuing) LinkedList has a constant time complexity.

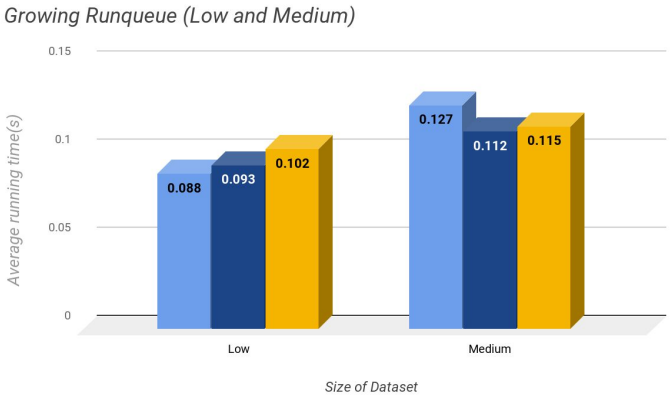
2. EMPIRICAL ANALYSIS

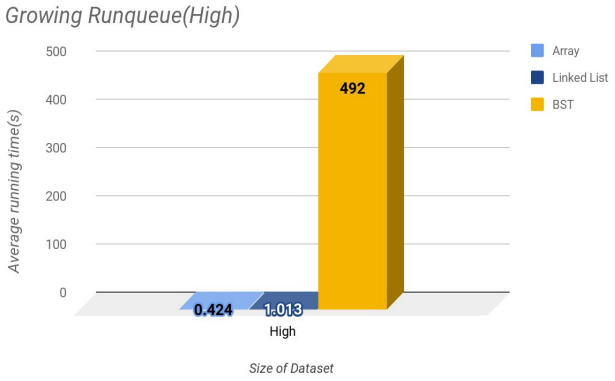
2.1 DATA GENERATION AND EXPERIMENTAL DESIGN

Data was generated with a random data generator using spreadsheets for the given three scenarios: growing runqueue, shrinking runqueue and calculating total vruntime of the preceding processes in three different sizes (Low, Medium and High). The processes in low settings have 100 counts, 1000 in medium size and 10000 in large size of the dataset. For each size setting, 3 samples were experimented with the three data structures and then average run time(in seconds) for each data structure was computed. By using Unix time command the time values were returned by the program as elapsed (real) time between invocation of algorithms, the user CPU time and the system CPU time. The average running times were calculated using spreadsheets.

2.2 EVALUATION

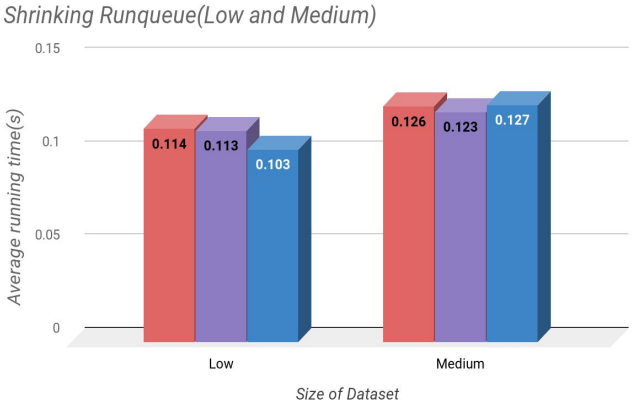
2.2.1 Growing Runqueue (Enqueue [EN])

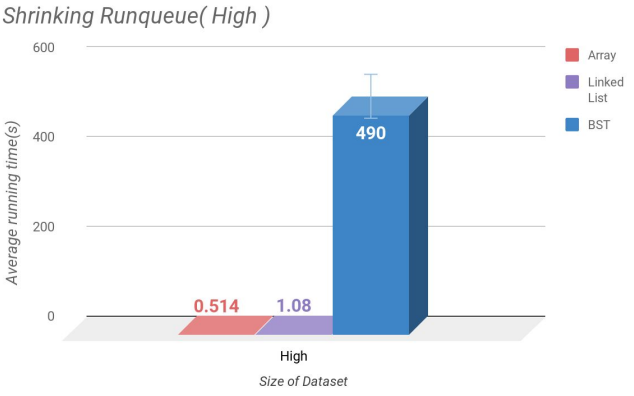




In the above illustration depicts that when in the growing order of the enqueue, all data structure worked differently in respect to average time with different sizes of operations. Comparatively, Array is slow when we take a medium number of processes to enqueue, whereas for the same data sets it took approximately the same time for linked list and binary search tree. Significantly, BST shows a very low performance for larger datasets, while ordered array showing high efficiency for larger datasets.

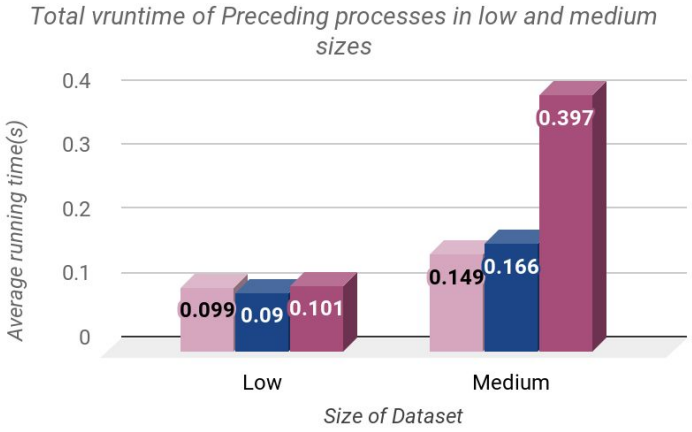
2.2.2 Shrinking Runqueue (Dequeue [DE])

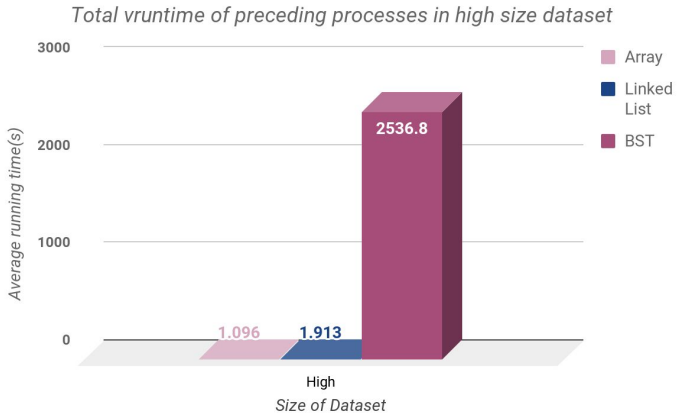




For dequeue operation, on running a low dataset BST turned out to be an efficient data structure with only a low significance. However, performance of BST for larger datasets is significantly very low. Furthermore, array and LinkedList are alike in terms of performance for every dataset. While evaluating the dequeue operation, we first fully-fledged the runque and then tracked the runtime of shrinking .

2.2.3 Total vruntime of preceding processes [PT]





Tabel representation of the above graphs :

Data Sets	Array	LinkedList	Binary Search Tree
Low	0.099s	0.09s	0.101s
Medium	0.149s	0.166s	0.397s
High	1.096s	1.913s	2536.8s

Total vruntime of preceding processes [PT]

The PT operation average running time was greater in comparison to the other operations , in this scenario first we flooded the runqueue with enqueue operations and then operated PT for those existing processes in the runqueue. For all datasets, array and linked list performed somewhat in a similar way, and similar to the previous two scenarios ordered array data structure shows high efficiency when data size increases. In contrast, when size of datasets increases, time efficiency of BST is exponentially decreasing, taking an average time of approximately 42 minutes to compute pt of 10,000 dataset.

3. RECOMMENDATIONS

Binary Search tree:

If the enqueue method is updated to create an AVL Tree (self-balancing Binary Search Tree), with additional property that difference between height of left subtree and right subtree of any node not to be more than 1 for all nodes, it is possible to improve the time complexity of the 3 scenarios from $O(n)$ to $O(log_2 n)$ - upper bound worst case scenarios.

Hence, we recommend an AVL tree (modified BST) as a better data structure for the Process scheduler.

4. SUMMARY

In conclusion to this Analysis , we have figured out that different data structures devour different running time and complexities based on the implementation. For the growing runqueue scenario, we recommend to use array for all size datasets because it's quick. Correspondingly for shrinking runqueue scenario LinkedList is best for low size datasets empirically, and furthermore theoretically it has constant time complexity. And, array is best for high setting datasets, and on the other hand, for calculating total vruntime we endorse LinkedList for less number of processes and array for medium and high number of processes. The reason for recommending particular data structures for particular datasets is that while tracking the average running time, as a result they appeared to be the fastest among others and efficient. Even though, in theory, when changing the runqueue, copying the array increases the time complexity, empirically it doesn't harm the performance of the ordered array. Overall we have the conclusion as :

1. In every Scenario for all data sets, array data structure has performed very consistently.
2. Ordered LinkedList and ordered array perform approximately similarly.
3. Binary Search Tree is the poorest data structure when it comes to very large datasets.

5. APPENDICES

In the submission file we included the following

I. Contribution Sheet.
II. Source code (6 java files).
III. Data Generation file.